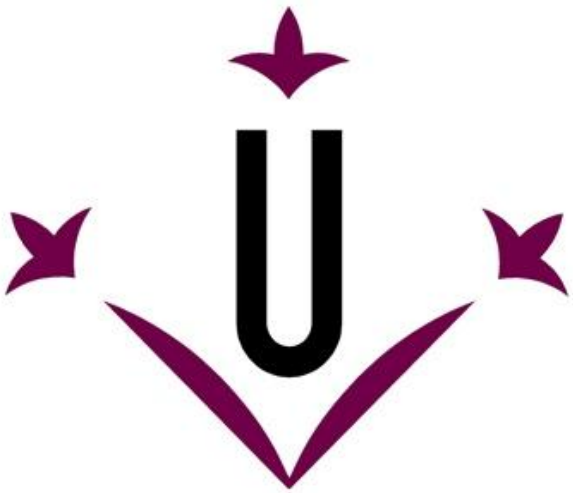

Universitat de Lleida

Escola Politècnica Superior

Enginyeria Tècnica en Informàtica de Sistemes

Projecte Final de Carrera

Disseny i implementació d'una API en Python per el processament de models de poblacions Simulats mitjançant PlinguaCore

Autor: Marc Trigueros Quibus
Director: Josep Lluis Lérida Monsó.
Juliol 2012

INDEX

INDEX DE FIGURES

INDEX DE TAULES

Agraïments

En primer lloc m'agradaria donar les gràcies al Sr. Josep Lluís Lérida Monsó, ja que sense la seva direcció, dedicació i motivació aquest treball no s'hauria pogut dur a terme.

Al Sr. Francesc Solsona Tehas i la Sra. Maria Angeles que en les successives reunions m'ha guiat en l'execució del projecte, en la definició dels requeriments de la solució i en l'aclariment dels nombrosos dubtes que han anat sorgint.

I també cal recordar, a la família, novia i amics per la seva paciència i recolzament en la realització del treball.

Resum

En aquest projecte, tractarem de crear una aplicació que ens permeti d'una forma ràpida i eficient, el processament dels resultats obtinguts per un eina de simulació d'ecosistemes naturals anomenada PlinguaCore [5] .

L'objectiu d'aquest tractament és doble. En primer lloc, dissenyar una API que ens permeti de forma eficient processar la gran quantitat de dades generades per simulador d'ecosistemes PlinguaCore. En segon lloc, fer que aquesta API es pugui integrar en altres aplicacions, tant de tractament de dades, com de cal·libració dels models.

El PlinguaCore, és un motor de simulació de models naturals escrits en llenguatge P-Lingua[6]. Aquest llenguatge, desenvolupat pel grup d'investigació natural de la Universitat de Sevilla, esta dissenyat per especificar una variant de la última generació de models computacionals definida per George Päun [7], anomenada P-Sistema. Els P-Sistemes[8] son uns tipus de models computacionals basats en el funcionament intern de les cèl·lules biològiques que mitjançant un modelatge adequat ens permet simular el funcionament de determinats processos naturals. Concretament la variant que ens interessa en aquest projecte es l'anomenada *Populatium Dynamic P-System model (PDP)*, i s'utilitza per la modelització de models poblacionals.

Tot i l'eficiència demostrada per aquest tipus de models en la modelització d'ecosistemes naturals, és molt difícil pels usuaris experts definir concretament els paràmetres que permet ajustar el model d'un ecosistema a la realitat. Per aquest motiu han de realitzar una gran quantitat de proves i processar-ne els seus resultats. La informació generada per aquest sistemes és molt gran i s'ha d'executar diferents proves d'un mateix ecosistema per normalitzar els resultats, i a més a més, els resultats generats s'han de processar de forma conjunta per ajustar el model.

Un cop processades les simulacions cal extreure els resultats i per això és necessari el disseny d'una API que ens permeti extreure'n la informació i fer els càlculs estadístics necessaris per validar el model, donar informació útil a l'usuari expert, etc.

Aquest projecte s'ha centrat en el disseny d'una API per l'extracció i processament dels resultats segons el requeriment d'un usuari expert. Aquesta API s'encarrega de retornar un valor anomenat error, aquest valor li servirà a l'usuari expert, per decidir si l'ecosistema que esta tractant es el que més s'aproxima a les dades reals.

Finalment, analitzarem l'eficiència en temps de còmput de les funcionalitats implementades i la quantitat de recursos que consumeix un cop executem la nostra API de càlcul d'ecosistemes.

Capítol 1

1 Introducció

Informalment, un model d'un cert fenomen, sistema o procés de la vida real és una representació concreta, abstracta, conceptual, gràfica o formal, que ens permet analitzar, descriure, explicar i, en general, aprofundir en el coneixement del mateix.

Els científics utilitzen regularment abstraccions de la realitat com diagrames, grafs, lleis, relacions, etc; amb l'objectiu de tractar d'entendre millor la realitat que examinen. En aquest context, les Matemàtiques i la Informàtica han estat utilitzades per aquests col·lectius com eines auxiliars per al millor desenvolupament d'aquests experiments.

Els grans progressos obtinguts durant el final del segle passat, amb les disciplines de Biologia cel·lular i molecular com amb Ecologia i no cal dir, amb Ciències de la computació, han propiciat l'avanç qualitatiu sobre els resultats obtinguts experimentalment, anomenats models formals.

Els models formals són abstraccions del món real dintre d'un marc matemàtic-computacional. És a dir, és una traducció de la realitat a un nou sistema expressat amb termes matemàtics-computacionals.

Concretament, nosaltres treballem amb un tipus de models computacionals que són els P-Sistemas. Aquesta tècnica de modelatge està inspirada en el funcionament intern d'una cèl·lula biològica; amb els seus orgànuls i la seva membrana.

En el present treball partim d'un model formal probabilístic, escrit en llenguatge P-Lingua[6]. Aquest model es pot aplicar a qualsevol tipus d'ecosistema, sempre i quant apliquem les adaptacions pertinents. S'ha utilitzat aquest P-Sistemes amb resultats força bons en l'estudi del trencalòs dels Pirineus [9][10][11][12], del mol·lusc zebra a l'embassament de Ribarroja o del Tritó Pirinenc [13], que es el que s'utilitza en aquest projecte com a model de referència.

Els ingredients bàsics d'un P-Sistema[8] són 3: l'estructura de membranes, el conjunt de regles que s'apliquen per evolucionar el model i l'alfabet d'objectes que es troben en les membranes. Les membranes representen els conjunts dins del model. Les regles son els patrons d'evolució del sistema, i els objectes son els elements que trobem en el model.

Aquest model formal quedarà encapsulat en un arxiu, el qual contindrà les regles del nostre ecosistema i els valors dels paràmetres, tan estàtics com probabilístics del model en qüestió. Un cop definits el model i els valors, es simula el ecosistema amb PlinguaCore i es comparen els resultats obtinguts amb els valors que es disposen de les experiències reals. Aquest procés de comparació ens servirà com a validació del model que estem creant.

Per tal de definir els paràmetres més adequats de les variables probabilístiques, els usuaris ecòlegs necessiten llençar gran quantitat d'experiments amb diferents paràmetres, per tal d'ajustar aquells valors, que proporcionen resultats el més semblants possibles a la realitat.

Aquestes execucions poden arribar a ser molt nombroses, i en ocasions molts costoses en temps. Això produeix un temps de calibratge molts cops inabordable, si volem analitzar totes les possibles combinacions de valors en els paràmetres que podria adoptar el nostre ecosistema.

Tot aquest càlcul té un cost molt elevat, computacionalment parlant. Aquí és on entra en joc la utilització d'entorns computacionals distribuïts, encarregats de presta'ns la capacitat de còmput necessària, per realitzar el calibratge d'un ecosistema, realitzant múltiples execucions paral·leles en un temps raonable per el usuari expert.

Un cop executats tots els experiments necessaris, cal extreure'n la informació especifica que l'usuari expert desitja. Això implica que l'usuari expert ha de poder definir quanta informació vol obtenir i de quantes espècies.

Aleshores s'ha de dotar el sistema d'un conjunt d'aplicatius capaços de processar i extreure la informació segons el que l'usuari expert desitja obtenir.

És aleshores, quan entra en joc el mòdul de càlcul d'ecosistemes, implementat en el projecte, encarregat de processar gran quantitat de simulacions i calcular quina d'aquestes simulacions del escenari, es la que s'aproxima més al comportament real de l'ecosistema.

1.1 PlinguaCore

En el present treball, treballem amb un simulador d'ecosistemes definit mitjançant un P-Sistema. Un P-Sistema[8] consisteix, en un model teòric de computació, inspirat amb el funcionament de les cèl·lules vives. Els components bàsics d'aquest tipus de simuladors P-Sistema són: La divisió del sistema per membranes, l'alfabet de treball i les regles d'evolució introduïdes per l'expert. Existeixen diferents variants de P-Sistemes, en el present treball ens hem centrat amb la variant que s'utilitza per moldejar l'estudi de poblacions, denominat *Population Dynamic P-System model (PDP)*.

Els paràmetres utilitzats per el model d'un cas concret, son sempre els mateixos, independentment del camí seguit per obtenir els resultats de sortida. Els paràmetres són les mostres extretes en el camp de forma empírica per els ecòlegs. Cal dir, que els valors obtinguts dels paràmetres, molts cops no són valors exactes, sinó que estem parlant de intervals de valors. Es per aquest motiu, que estem obligats a calibrar el model. Això significa, buscar els valors dels paràmetres que més s'aproximen a la realitat observada per l'expert.

En la Figura 1 podem observar una estructura de membranes PDP. A l'interior de l'entorn o pell veiem les diferents membranes que conformem el P-Sistema, que no és més que un conjunt de membranes jerarquitzades carregades elèctricament. A l'interior de les membranes, podem veure regions on trobaríem els objectes de l'espècie a estudiar per l'expert (menjar, animals, condicions mediambientals, reproducció, etc...). Aquest objectes aniran evolucionant, mitjançant les regles d'evolució fixades per l'expert, podent moure's els objectes entre les diferents membranes, sortir de l'entorn o generar nous objectes.

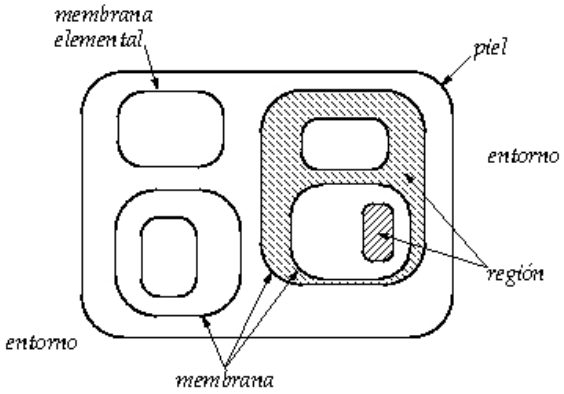

Figura 1: Estructura de membrana

En el nostre treball l'encarregat de modelar aquest P-Sistemes és el PlinguaCore, estem parlant d'un simulador d'un model formal de computació a una aplicació software que descriu les especificacions de l'ecosistema a través d'un llenguatge de programació i captura la semàntica mitjançant d'un algoritme de simulació que té de reproduir l'ecosistema amb fidelitat; és a dir, cada pas de computació del model formal és reproduït en el simulador a través d'un número finit de passos, d'aquesta forma el simulador es capaç de determinar l'evolució dels elements bàsics del model que han tingut lloc en cada pas.

En els models PDP les regles d'evolució que s'apliquen compleixen les característiques de la Figura 2.

$$r \equiv u^a[v^b]_i^\alpha \xrightarrow{f_r} u'^c[v'^d]_i^{\alpha'}$$

Figura 2: Regles d'evolució en el model PDP

Podem veure exterior a la membrana "i", que posseeix carrega "α" a l'objecte "u" amb multiplicitat "ɑ ". A la membrana etiquetada amb el caràcter "i", trobem l'objecte "υ" amb multiplicitat "b", amb una probabilitat "f_r " que s'aplicarà la regla de manera que modificarà la càrrega de "α" a "α'" i els objectes "u i υ" evolucionaran a "u' i υ'" , amb multiplicitat "c i d".

El nostre simulador d'ecosistemes tindrà els elements comuns d'un P-Sistema que mostra el diagrama de flux de la Figura 3.

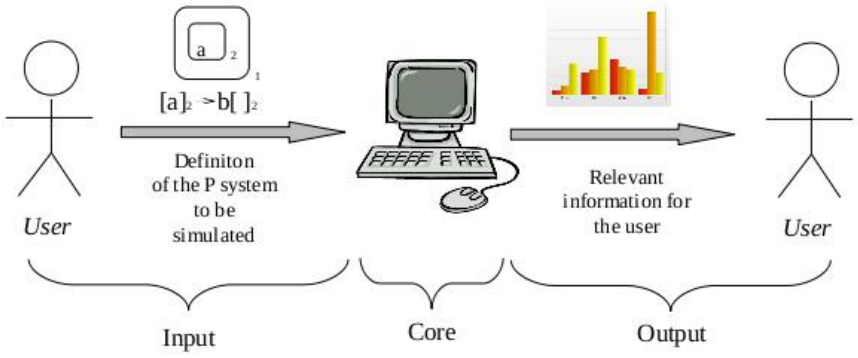

Figura 3: Elements comuns en els simuladors P-Sistema

El primer pas és decidir quina serà la definició del model que s'ha de simular, en el nostre cas el model PDP d'evolució. Posteriorment necessitarem un nucli de computació, en el nostre cas el PlinguaCore[5], encarregat de simular l'ecosistema tants cops com l'expert decideixi. Per acabar, presentarem els resultats obtinguts a l'usuari expert en format d'arxius, on per a cada arxiu tindrem com a punt de partida la configuració inicial formada per l'estructura de membranes, les regles d'evolució i l'alfabet inicial. Per cada pas de computació s'aplicaran totes les regles possibles obtenint una nova configuració, la seqüència de configuracions formarà una simulació.

Quan s'ha de fer un gran nombre d'experiments i es vol estalviar temps, és important recorre a noves solucions per resoldre l'execució de les simulacions. Aquí és on entra la utilització de sistemes distribuïts, que permetrà l'execució paral·lela d'un gran nombre de simulacions realitzades per el PlinguaCore[5].

A més a més, necessitarem d'un sistema que gestioni els resultats i els processi de forma conjunta. Es aquí, on incorporaríem el sistema de càlcul desenvolupat en el present projecte, per validar els experiments i poder certificar quina de les configuracions de paràmetres ajusta millor el comportament del model al sistema real observat.

1.2 Sistema de Calibratge

Definirem com a sistema de calibratge, al procés per el qual es determinen o s'ajusten els paràmetres d'entrada, introduïts per l'usuari expert com a conseqüència de realitzar estudis empírics en el medi, per poder generar diferents simulacions d'un mateix ecosistema.

En el present treball, serà de gran importància el procés de calibratge, ja que els paràmetres que l'expert necessita introduir en el model d'estudi d'un ecosistema, esta determinat per les mostres extretes empíricament en el medi i en molts casos alguns d'aquest paràmetres no es pot donar un valor concret, ja sigui per la dificultat en obtenir la informació, per la falta d'observacions, per la falta de dades o errors en la presa d'informació, etc. En aquests casos els usuaris experts determinen uns intervals de valors que poden prendre aquests paràmetres.

És per aquest motiu que ens és de gran importància el calibratge, ja que l'ús de la API implementada en aquest projecte, per a l'extracció i el processament de la informació, ens permet ajustar els valors dels paràmetres que fan que el model reprodueixi més fidelment la realitat.

1.2.1 Calibratge del model i verificació

En el nostre projecte, per obtenir un bon calibratge, serà necessari ajustar els paràmetres introduïts per l'expert el millor possible. Per poder-ho aconseguir, l'expert ha de generar múltiples escenaris amb diferents valors per els paràmetres d'entrada. Definim escenari, com una instancia o assignació de valors als paràmetres de entre totes les combinacions posibles. Un cop s'ha generat els N escenaris serà necessari simular cada escenari varies vegades i comparar els resultats obtinguts amb els valors observats reals. Tot aquest procés té la finalitat d'obtenir l'escenari que millor s'apropa al comportament real.

De la simulació dels escenaris obtindrem un arxiu com el que mostra la Figura 4, el qual ens mostrarà el nom que li dona l'expert a l'escenari, a més a més, incorporarà de forma visible, el nombre de paràmetres d'entrada que l'expert a variat per obtenir aquest escenari.

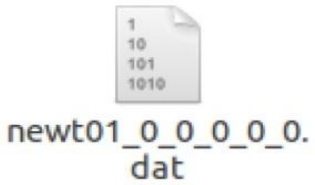

Figura 4: Escenari generat

Cal dir, que l'obtenció d'aquest escenari és molt costosa, ja que en moltes ocasions li és molt complicat a l'expert acotar els paràmetres d'entrada, això implica que ens trobem davant d'un problema combinatori amb un cost computacional que pot arribar a ser molt elevat.

Segons els estudis realitzats en el article *Paralelización de un algoritmo de calibrado para el moldeado de ecosistemas* [14], si suposem que estem treballant amb 3 paràmetres que poden prendre 5 valors possibles, podem generar fins a 5^3 possibles escenaris. En el cas d'utilitzar 30 simulacions per normalitzar cada un dels escenaris, i suposant que triga en generar cada simulació sobre 1 minut i mig, requeriríem ($5^3 * 30 * 1.5 \frac{minuts}{simulació} = 5625\, min.$) aproximadament 4 dies per ajustar els paràmetres dels tres escenaris als valors que millor resultats ens donen.

Un cop obtenim els escenaris, caldrà realitzar el procés de calibratge. D'aquest procés s'encarregarà l'API de càlcul implementada en aquest projecte. Els resultats de cada escenari són comparats amb uns valors de referència proporcionats per l'usuari expert provinents d'observacions reals.

1.3 Llenguatges d'scripting

En l'àmbit de la informàtica un guió, arxiu d'ordres o arxiu de processament per lots, vulgarment referit amb el barbarisme script, és un programa format per un conjunt seqüencial d'ordes simples, que molts cops és emmagatzemat en un arxiu de text pla. Els guions o scripts són quasi sempre interpretats, és a dir, el conjunt d'instruccions s'executen una a una en ordre seqüencial a diferència dels llenguatges compilats en els que el compilador s'encarrega de fer la traducció de tot el codi font i un cop traduït crearà un executable amb codi màquina associat a un determinat computador .

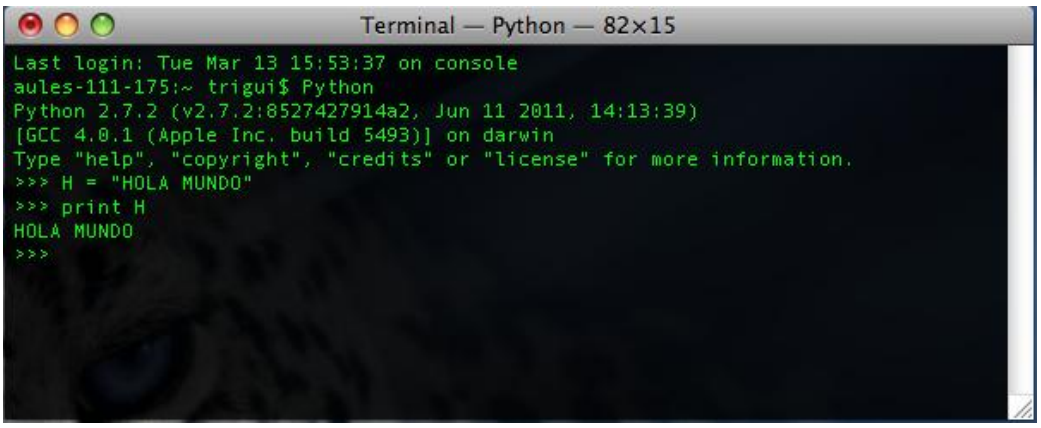

Figura 5: Exemple bàsic d'script o Guió

Els avantatges d'utilitzar un llenguatge d'scripting en el nostre projecte, és que permet interactuar de forma ràpida amb el sistema operatiu, permet automatitzar fàcilment les tasques de manteniment i resulta molt adequat a l'hora de manipular arxius de text, com els retornats per les simulacions amb el PlinguCore[5], a més a més, els scripts son multiplataforma, podem executar aquests scripts tant en Windows, Linux o Mac Os.

1.3.1 Classificació dels llenguatges d'scripting

Per poder realitzar aquest projecte, hem tingut de decidir si utilitzaríem un llenguatge d'scripting **Estructurat** o requeriríem d'un llenguatge d'scripting **Orientats a objectes**. A continuació podem veure les característiques tant d'un com del altre, i així poder prendre la decisió més adequada segons les nostres necessitats.

Estructurats:

A finals del 1970 va sorgir una nova forma de programar que no solament donava lloc a programes fiables i eficients sinó que a més a més estaven escrits de manera que facilitava la seva comprensió.

La programació Estructurada o No Orientada a objectes, és una tècnica per escriure programes de forma clara. Per poder realitzar aquest programes s'utilitzen únicament tres tipus de estructures diferents: Seqüència, selecció i iteració.

- **Seqüència:** Execució d'una instrucció darrere de l'altra.
- **Selecció:** Execució de una de les dues instruccions (o conjunts), segons el valors de una variable booleana.
- **Iteració:** Execució de una instrucció (o conjunt) mentre una variable booleana sigui verdadera. Aquesta estructura lògica s'anomena bucle.

Tan sols amb aquestes tres estructures és poden descriure tots els programes i aplicacions possibles. Si més no, els llenguatges de programació tenen un major repertori d'estructures de control, aquestes poden ser construïdes mitjançant les tres bàsiques citades.

Son definits com a llenguatges estructurats: **Bash**, **sh** o **csh**

Orientats a Objectes:

L'orientació a objectes és un paradigma de la programació en que tot és basa en objectes. Un objectes és un tipus abstracte de dades que encapsula tant les dades necessàries com les funcions per accedir-hi.

L'orientació a objectes ofereix millores molt significatives en el disseny, desenvolupament i manteniment del software oferint una solució a llarg plaç dels problemes i preocupacions que han existit desde els inicis en el desenvolupament de software.

Aquest problemes han sigut: la falta de portabilitat del codi i la seva reutilització, codi que és difícil de modificar, processos de desenvolupament llargs i tècniques de codificació no intuïtives.

Un llenguatge orientat a objectes ataca aquest problemes. Té tres característiques bàsiques: a d'estar basat amb objectes, ha d'utilitzar classes i ha de ser capaç de proporcionar herència de classes.

En la Figura 6 podem veure les principals característiques que formen el llenguatge orientat a objectes, són característiques indispensables per poder dir que estem treballant amb el POO.

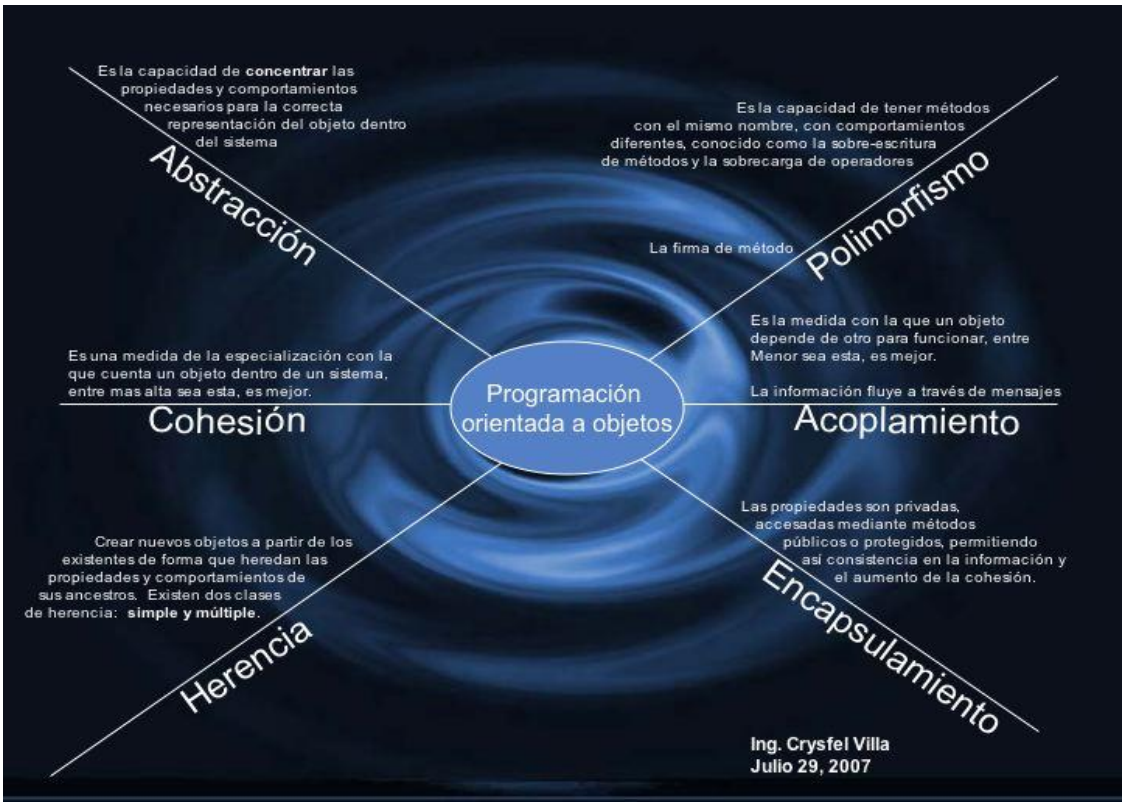

Figura 6: Com està format el POO.

1.3.2 Tecnologies utilitzades

Per poder realitzar el present treball, hem escollit com a llenguatge de programació Python[1], ja que és un llenguatge d'scripting orientat a objectes destacat per la seva simplicitat, potència i la velocitat a l'hora de desenvolupar aplicatius.

Des de l'inici, tant a mi com al director del projecte, ens va quedar clar que era imprescindible la utilització d'un llenguatge d'alt nivell orientat a objectes, ja que els llenguatges estructurats no són els més adequats per realitzar processos d'una certa complexitat; treballar amb operacions internes sobre llistes, la implementació d'un Hash o ajudar-nos de patrons per realitzar cerques complexes.

Cal dir, que a diferència dels llenguatges estructurats, la utilització d'un llenguatge orientat a objectes m'ha simplificat la feina, al tenir incorporat mòduls que posteriorment han estat cridats en el programa sense la necessitat de ser implementats desde zero per mi, i el que es més important s'han pogut reutilitzar funcions i no ha estat necessari tornar-les a implementar.

1.3.3 Perquè Python?

No és un requisit imprescindible la utilització de Python com a llenguatge de programació per l'API de processament d'ecosistemes, ja que existeixen d'altres, que possiblement podrien haver servit per poder fer el mateix tipus de procés, com podria ser: **Ruby**[16], **Perl**[17], **Groovy**[18] o **Lua**[19].

En el present projecte, tenim la problemàtica que estem treballant amb molta informació repartida per múltiples arxius de text pla generats per el PlinguaCore[5], es necessari processar'ls un per un, i a més a més, cal rebutjar de cada arxius generat per el PlinguaCore la informació redundant per l'usuari expert a l'hora de prendre decisions. Python[1] és una bona opció ja que la seva sintaxis i semàntica senzilla ens ajudarà a realitzar tot aquest procés de parseig, filtratge i obtenció de resultats, ja que té múltiples eines per processar aquests arxius d'una forma ràpida i eficient, gràcies la seva gran biblioteca de classes, la qual subministra tot tipus de operacions especifiques sobre llistes, strings, patrons, etc, que ens a simplificat el codi, l'ha fet molt més intuïtiu, ens ha alliberat d'informació redundant, ens ha generat arxius de forma molt ràpida i hem obtingut una execució ràpida dels script implementats.

Però un dels factors decisius a l'hora de prendre la decisió d'utilitzar Python ha estat, l'experiència que té la EPS amb Python, ja que en anteriors projectes de similar característiques s'ha utilitzat el llenguatge esmentat, amb resultats molt bons i eficients, també cal indicar, que en la enginyeria superior és un llenguatge d'scripting utilitzat i podria tenir suport en el cas de troba'm amb dificultats a l'hora de realitzar la implementació del codi.

1.4 Objectius generals

Els usuaris experts tenen la necessitat d'obtenir un model formal o matemàtic-computacional de les simulacions dels escenaris que estem estudiant, per així poder prendre decisions sobre el medi d'una forma ràpida i el més encertada possible.

En el present treball, disposem d'un simulador d'ecosistemes descrits mitjançant models PDP, a més a més, disposem d'un model d'ecosistema natural, en el nostre cas concret processarem el model del Tritó Pirinenc[13]. El Tritó, és una espècie endèmica que habita als Pirineus i el Pre-Pirineu. El model es centrarà en el torrent del Vall del Pi que forma part d'un conjunt de 38 torrents, 16 dels quals estan ocupats per el Tritó. Proper a aquest torrent, trobarem 8 torrents més que també influeixen en la seva dinàmica i que, per tant, hauran de ser inclosos en el model.

Per poder tractar aquest model natural, l'usuari expert generarà múltiples escenaris amb diferents valors en els paràmetres, que simularan els possibles condicions que pot adoptar l'ecosistema. Un cop obtenim els diferents arxius amb les simulacions de cada escenari, utilitzarem la nostra API implementada en aquest projecte per al seu processament. Aquesta mateixa API, incorporarà una funció per poder comparar els resultats obtinguts gràcies els nostre simulador d'ecosistemes amb els valors obtinguts en el ecosistema real.

Capítol 2

2. Anàlisis de solucions

Python és un llenguatge de programació modern creat per Guido van Rossum a principis del any noranta. Es tracta d'un llenguatge d'scripting independent de plataforma i orientat a objectes, preparat per realitzar qualsevol tipus de programa, desde aplicacions Windows a servidors de red o inclús pàgines web. Es un llenguatge interpretat, el que significa que no necessita compilar el codi per poder executa'l, això provoca avantatges com la gran velocitat de desenvolupament però també inconvenients, si intentem implementar codis molt extensos no seran molt ràpids. Es de codi lliure i es pot descarregar desde el lloc oficial[1]. Que sigui una tecnologia oberta i lliure té una sèrie d'avantatges importants sobre les tecnologies propietàries, la principal es que és pot utilitzar sense cobrir costos de llicència.

Les característiques del llenguatge les podem agrupar de la següent forma:

- **Propòsit general**:

  Python permet crear tot tipus de programes, és a dir, no es un llenguatge centrat solament en un propòsit, hi ha grans empreses com; Google, Yahoo, entre moltes més que utilitzen el llenguatge.

- **Multi plataforma:**

  Podem trobar-hi versions de Python per molts sistemes operatius: Windows, Mac Os, Unix, entre d'altres. Cal dir, originalment es va desenvolupar per a Unix, encara que qualsevol sistema es compatible sempre que en el sistema operatiu en qüestió tinguem un intèrpret instal·lat.

- **Interpretat:**

  Significa que no és necessari compilar el codi abans de la seva execució. El intèrpret es un programa capaç d'analitzar i executar altres programes escrits amb un llenguatge de programació d'alt nivell. La diferència més gran que hi ha entre un llenguatge interpretat i un compilat, es que en el compilat, el compilador fa la traducció total d'un programa a codi màquina del sistema, mentre que els interpretats sol realitzen la traducció a mesura que es

necessari, normalment instrucció a instrucció, i no guarden els resultats de la traducció.

- **Interactiu:**

  Python disposa d'un intèrpret per línea de comandes, en el que es poden introduir sentències. Cada sentència s'executa i produeix un resultat visible, el qual ens pot ajudar a entendre millor el llenguatge i provar els resultats de les execucions de porcions de codi, d'un forma molt ràpida.

- **Orientat a objectes:**

  La programació orientada a objectes es suportada per Python i ofereix en moltes ocasions una forma senzilla de desenvolupar programes amb components reutilitzables.

- **Sintaxis:**

  Cal destacar, que Python té una sintaxis molt visual, gràcies a una anotació identada (amb marges). En molts llenguatges, per separar porcions de codi, s'utilitza elements com les claus o paraules clau "begin" i "end". Per separar les posicions de codi a Python s'ha de realitzar tabulacions.

Totes aquest ampli ventall de característiques, fan que aquest llenguatge sigui el que em escollit per desenvolupar el present projecte. En la secció següent, descriurem les parts més importants d'aquest llenguatge, utilitzades en la implementació del projecte.

2.1 Llibreries de Python per aquest projecte

Per poder realitzar aquest projecte hem tingut de treballar amb objectes i funcions Python que ens han facilitat molt la feina a l'hora de realitzar la implementació del processament. Per poder tenir un bon seguiment del procés d'implementació és necessari fer una guia on és recull els objectes i funcions implementats, i on és detalla el seu funcionament per a futures ampliacions o modificacions dels scripts implementats en l'estudi dels ecosistemes.

Inicialment explicarem les funcions utilitzades del mòdul Os de Python, aquest mòdul esta composat per desenes de funcions, que interactuen amb el sistema operatiu, entre les quals cal destacar les següents:

- **Abspath:** La funció ens retornarà la ruta absoluta en forma d'string d'un arixu o d'una carpeta en el que volem realitzar algun procés, per després poder treballar amb ella.

  Un exemple d'ús, on retornem la ruta absoluta d'un arxiu, seria el següent:

  >>>Ruta=os.path(TFC_Marc.doc)

  >>>Ruta

  >>> /home/marc/TFC_Marc.doc, retornarà la ruta completa de l'arxiu.

- **listdir(path) :** La funció ens retornarà una llista amb les carpetes o arxius que podem trobar en la ruta que li hem passat.

  Un exemple de crida podria estar el següent:

  >>>Lst = os.listdir(path), on Lst és la llista contenidora, on emmagatzemem el que ens retorna la funció.

  >>>Si en el directori path tenim dos arxius el resultat seria el següent: [Arxiu_0, Arxiu_1]

- **mkdir(path) :** La funció crearà en el directori "path" una carpeta.

  Un exemple d'ús seria el següent:

  >>>path = "/home/marc/TFC_Marc"

  >>>os.mkdir(path)

  D'aquesta forma, en el directori passat per el path tindrem la nova carpeta TFC_Marc.

- **spawnv:** Aquesta funció ens permet treballar amb l'execució d'scripts, ja que si tenim de treballar amb varis scripts que s'han de executar un després de l'altre d'una forma seqüencial per tal que no finalitzi el bucle d'execució fins que no s'hagin executat tots els scripts s'ha d'utilitzar aquesta funció, que consta de tres paràmetres d'entrada. El primer és el mode d'execució, aquí és on notificarem a la funció que a d'esperar a concloure el bucle, El segon paràmetre és l'script a executar i per últim una llista amb els paràmetres necessaris per executar l'script amb el que estem treballant.

Cal destacar també la utilització del mòdul Sys de Python, aquest mòdul permet l'accés a algunes variables utilitzades per el shell i funcions que interactuen amb el shell.

- **argv[i+1]:** La funció ens permet passar-li arguments al nostre arxiu Python per després poder-los utilitzar. La forma de passar els arguments, és per el Shell. Quan parlem de argv[0] ens referim al nom d'arxiu que estem executant, argv[1] serà el següent paràmetre introduït i així successivament, d'aquesta forma, gràcies a la interactuació amb el shell, podem utilitzar els paràmetres introduïts, per després utilitzar-los en els nostres scripts.

Han estat de gran importància les funcions que treballen sobre llistes, ja que en tot el tractament i parseig implementat en el projecte, s'ha requerit l'ús d'aquestes funcions. A continuació mostraré unes de les més importants:

- **Strip:** És una operació que ens permet manipular el contingut d'un string. Donat un string el que farà es eliminar els espais en blanc que pugui haver-hi.

  Un exemple de la crida podria ser el següent:

  Si tenim l'string "L= Hola món" i utilitzem la funció "Strip" sobre l'string, obtindrem la cadena de caràcters sense espais en blanc.

  >>>L.split()

  >>>Holamón

- **Split:** La operació ens permet treballar amb molta facilitat amb strings. Utilitzarà un separador que podria ser ",", ".", ":", ";", etc... , segons li sigui més interessant al programador, aquesta funció convertirà l'string en una llista dividida en tantes fraccions com nombre de vegades a trobat el caràcter utilitzat com a separador.

  Un exemple de crida a la funció pot estar el següent:

  Si Tenim l'string "L = 1,2,3,4" i utilitzem la funció "Split" sobre l'string, obtenim els resultats mostrats que podem veure a continuació.

  >>>L.split()

  >>>[0, 1, 2, 3, 4] on L[0] = 1, L[1]= 2, L[2]= 3 i L[3]= 4.

- **Rsplit:** Aquesta operació ens permet treballar amb gran facilitat amb strings, el procés que seguirà és el següent: utilitzant un separador que podria ser ",", ".", ":", ";", etc... , segons li sigui més interessant al programador, la operació separarà l'string agafant el primer caràcter utilitzat com a separador situat a l'extrem més a la dreta.

  Un exemple de la crida a la funció pot ser el següent:

  Treballem amb l'string "L = newt01.0.1.avg", si realitzem l'operació sobre L, L.rsplit() obtindrem una llista dividida en dos posicions on L[0]= newt01.0.1 i L[1]= avg.

- **Append:** És una operació que ens permet introduir en una llista un string, aquest procés es farà de forma molt senzilla.

  Un exemple de la crida a la funció pot ser el següent:

  Inicialment hem de declarar una llista buida, com per exemple Lst = []. En l'exemple treballarem amb l'string Str = Hola món, per carregar l'string a la llista ho farem: *lst1.append(string1).*

  De forma que lst1[0] = Hola món.

Per realitzar el present treball hem tingut de treballar amb gran quantitat d'arxius, així que s'han utilitzat funcions que treballen directament sobre fitxers. En aquest paràgraf mostro les que hem utilitzat.

- **F = open('arxiu','r/w/rw') :** Aquesta operació ens permet obrir un arxiu en el format que l'usuari programador li sigui més oportú. El camp "arxiu" com mostra de forma intuïtiva, es el nom del arxiu a obrir, i el segon camp marca com el volem obrir, si en "r" sol lectura, "w" escriptura o "rw" tant escriptura com lectura. La variable "F" és el objecte que utilitzarem quan volem treballar sobre el arxiu.
- **F.close()** : La funció ens permet tancar un arxiu després de llegir-lo o modificar-lo. Aquesta funció és molt important, ja que deixar els arxius oberts un cop hem acabat de treballar amb ells pot provocar errors i pèrdua d'eficiència en el nostre programa.
- **f.write(x)**: Funció que ens permet escriure en un arxiu "f" el contingut de la variable "x".

Una de les funcions més importants, per carregar les espècies i el nombre d'individus de cada espècie a estudiar per l'expert, són les operacions que treballen sobre un diccionari, el qual és una estructura composada per una clau i un valor, entres les quals destacarem les següents:

- **Dic{key,value}:** Aquesta operació ens permet crear un diccionari o matriu associativa, el diccionari esta composat per la tupla clau i valor, el que vull dir, és que per cada clau tenim un valor associat.

  Per exemple:

  >>>Dic = {"gener":1, "febrer":2, "març":3}

  >>>{'gener':1, 'febrer':2, 'març':3}

- **Dic[key]= value :** D'aquesta forma, carregem donat una clau el valor que volem que prengui aquesta clau.

  Per exemple:

  Si value = 5 i key = gener, obtindrem el següent resultat.

  >>>{'gener':5, 'febrer':2, 'març':3}

- **Dic[key]=0 :** D'aquesta forma assigno a una clau el valor "0". En el projecte s'utilitza en una estructura "for" per inicialitzar tot el diccionari a 0.

  Després de realitzar la funció obtindrem:

  >>>{'gener':0, 'febrer':0, 'març':0}

- **Dic.get(key):** Aquesta funció ens retornarà el valor al que esta associat la clau que li estem passant.

  Si la clau és febrer, el valor que en retornarà serà:

  >>> 2.

- **Dic.items():** Retornarà un llista de tuple amb claus-valors del nostre diccionari.

  La execució de la funció retorna el següent resultat:

  >>>{('gener':1), ('febrer':2), ('març':3)}

- **Dic.Key():** Retorna una llista de les claus que té el nostre diccionari.

  La execució de la funció retorna el següent resultat:

  >>>[‘gener’, ‘febrer’, ‘març’]

- **Dic.Values():** Retorna una llista dels valors que té cada clau el nostre diccionari.

  La execució de la funció retorna el següent resultat:

  >>>[‘1’, ‘2’, ‘3’]

A l’hora de realitzar cerques sobre strings, llistes o treballar amb patrons, ens ha estat necessari treballar amb el mòdul RE de Python, el qual s’encarrega d’utilitzar expressions regulars per poder realitzar els processos.

- **search(value, list) :** Aquesta funció ens permet fer una cerca en una llista. La variable “value” és el valor que volem cercar en la nostra llista i “list” és la llista en la que fem la cerca.

Quan hem necessitat extreure resultats estadístics, tant la mitjana aritmètica, desviació o diferència de quadrats, s’ha utilitzat el mòdul Math de python, encarregat de subministra’ns operacions matemàtiques que ens han agilitzat la feina.

- **Math.pow(value, 2) :** Aquest mòdul treballa amb operacions matemàtiques. En aquest cas el que ens retornarà es el quadrat del valor introduït en la variable “value”.

  Un exemple d’execució seria el següent:

  >>>math.pow(5,2)

  >>>25

- **math.sqrt(value) :** En aquest cas el que ens retornarà es l’arrel quadrada del valor introduït en la variable “value”.

  Un exemple d’execució seria el següent:

  >>>math.sqrt(25)

  >>>5

Capítol 3

3. Implementació de la API de processament de resultats

En el diagrama que mostrem en la Figura 7**,** es descriu el procediment de calibratge on integrarem l'API desenvolupada en aquest projecte. En aquest esquema podem observar dues etapes clarament diferenciades, la primera consisteix en l'obtenció de les simulacions d'un ecosistema_X i la següent serà la utilització de l'API de processament de resultats, on es processen els múltiples resultats de cada escenari i on es calcula l'error obtingut en la generació de l'ecosistema, respecte les dades reals proporcionades per l'usuari expert.

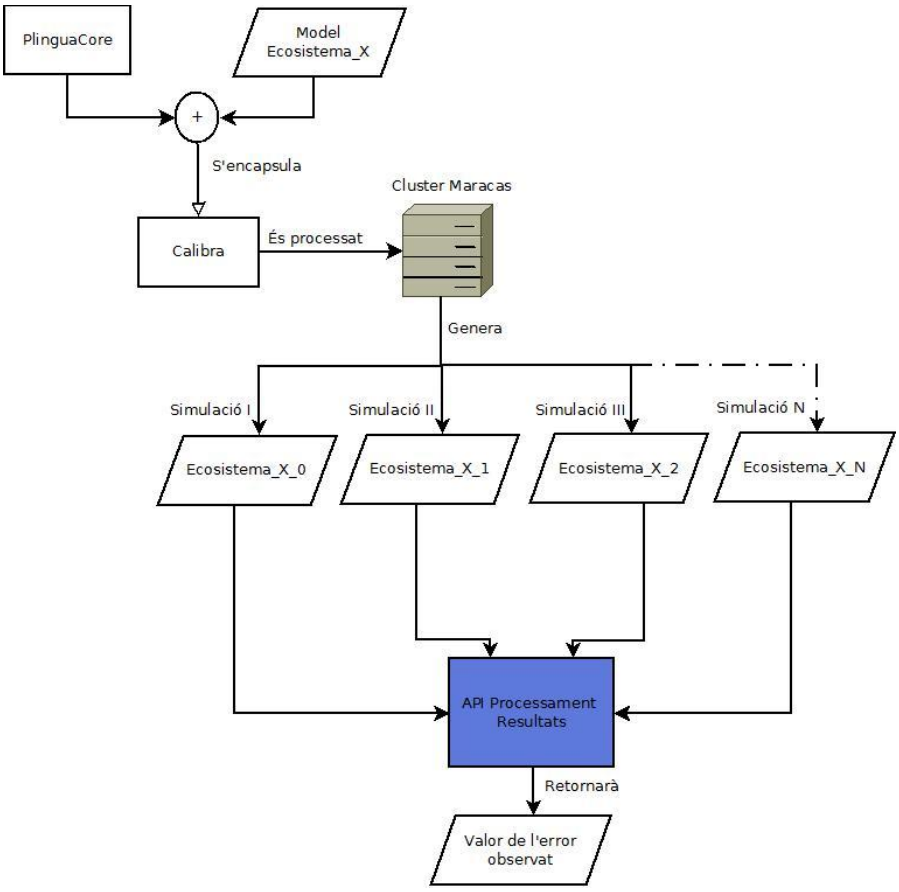

Figura 7: Diagrama de flux del projecte

A continuació descriurem breument en que consisteix cada etapa del procés de calibratge i on incideix les implementacions realitzades en la API de processament de resultats desenvolupada en aquest projecte.

- **Obtenció de les simulacions d'un escenari:**

  Per poder obtenir les simulacions d'un escenari, necessitem un sistema de còmput distribuït, en el nostre cas em utilitzat el clúster Maracas[4], el qual mitjançant un script en Bash[15] realitzarà les simulacions dels escenaris que volem processar. En el procés Calibra, definim els escenaris a processar, el nombre de simulacions que volem realitzar i el nombre de passos de simulació que volem que realitzi el PlinguaCore. Un cop finalitzat el procés de simulació, obtenim els arxius de sortida amb els resultats de les simulacions dels escenaris, Ecosistema_X_0, Ecosistema_X_1, ..., Ecosistema_X_N.

- **Obtenció de l'error en executar l'API implementada:**

  Un cop disposem dels resultats de les simulacions de cada escenari, es necessari implementar un sistema que pugui processar tota la informació generada pel PlinguaCore. El processament d'aquesta informació, implica: extreure la informació redundant abocada pel PlinguaCore deixant només la informació important per ser processada. Resumir per anys i espècies la informació en que l'usuari expert està interessat, a partir dels arxius obtinguts de la primera fase del procés. Realitzar els càlculs estadístics sobre les diferents simulacions i espècies determinades per l'usuari expert. Finalment es comparen els resultats obtinguts amb dades reals i s'obté el error produït pel ecosistema processat.

Posteriorment es descriuran els diferents scripts, amb les seves característiques, funcionament, llibreries Python necessaris per ser implementats i problemes que m'he trobat a l'hora d'implementa'ls.

3.1 Procés de simulació d'escenaris

El primer obstacle que ens apareix en el projecte, és automatitzar les simulacions dels escenaris, ja que necessitarem generar tantes simulacions com l'usuari expert indiqui. La necessitat de generar grans quantitats de simulacions, és perquè l'estadística ens diu que si prenem N valors d'un cert paràmetre d'estudi, el seu promig serà un valor que podem considerar com a correcte quan el nombre N sigui prou gran (N>30) .

Per poder realitzar les simulacions dels escenaris, em desenvolupat script programat en Bash que li direm **execplingua.sh**, el qual executarà el simulador PlinguaCore[5]. Per poder executar el PlinguaCore necessitarem els següents paràmetres d'entrada: directori on es troba l'escenari, directori destí on emmagatzemem les simulacions i el nombre de passos que harem de simular, on els passos es una unitat temporal (setmanes, mesos, anys, etc.), definida per l'usuari expert, amb la qual podem obtenir la informació necessària pel model que estem processant (en el nostre cas el Tritó).

Un cop tenim definits els paràmetres d'entrada del PlinguaCore, per poder obtenir resultats correctes, serà necessari generar múltiples simulacions d'un mateix escenari i realitzar el promig d'aquestes. Per realitzar aquest procés caldrà llençar múltiples tasques (una per cada simulació), a la unitat de còmput distribuït.

Per poder realitzar el procés de simulació, serà necessari implementar un nou script que ens simplifiqui el procés, el qual li direm **ecosystem.py**. Aquest script per poder ser executat requerirà dels següents paràmetres d'entrada, mostrats en la Figura 8:

- El primer paràmetre d'entrada és el nombre de simulacions que l'expert vol executar per escenari, precedit per l'argument ***–n***.
- El segon paràmetre d'entrada és el directori origen on trobem els escenaris que cal simular, precedit per l'argument ***–rs***.
- Per últim, el paràmetre d'entrada és el directori destí on emmagatzemarem les simulacions processades pel PlinguaCore, precedit per l'argument ***–ro***.

Aquest script l'encapsularem sobre un llançador d'escenaris anomenat **Calibra.sh**, mostrat en la Figura 8. Etiquetem com a llançador d'escenaris al **calibra.sh**, perquè es justament la seva funció, per cada ecosistema que l'expert vol estudiar, generarem un calibra i l'encuarem a la unitat de còmput distribuït com si fos una tasca més. Gràcies aquest procés, ens evitarem de generar múltiples tasques en el clúster, sol generarem una tasca per cada escenaris que es vulgui processar.

calibra_0.sh

```
#!/bin/bash

# Consola que es vol utilitzar
#$ -S /bin/bash
# Exportar totes les variables d'entorn
#$ -V
# Fixar el directori de treball al mateix desde on llanÃ§a el script
#$ -cwd


# EL QUE ES REALMENT IMPORTANT
# Quans processos es volen llanÃ§ar per treball o quanta memoria RAM es vol reservar
# (1 proces = 1G de RAM)
#$ -pe smp 4
# Limit de memoria RAM (OBLIGATORI!)
#$ -l h_vmem=1G
#$ -o /home/mtrigueros/output
#$ -e /home/mtrigueros/output

# Aqui   el que es vol executar

#./execplingua.sh

# Nova execució

./ecosystem.py -n 100 -rs plifiles/newt01_0_0.pli -ro datfiles

```

Figura 8: Arguments d'execució ecosystem.py

En la Figura 9 podem veure el diagrama de flux del procés d'obtenció de les simulacions per escenari. En l'exemple podem veure, l'obtenció de les simulacions de tres escenaris (Escenari_A, Escenari_B i Escenari_N).

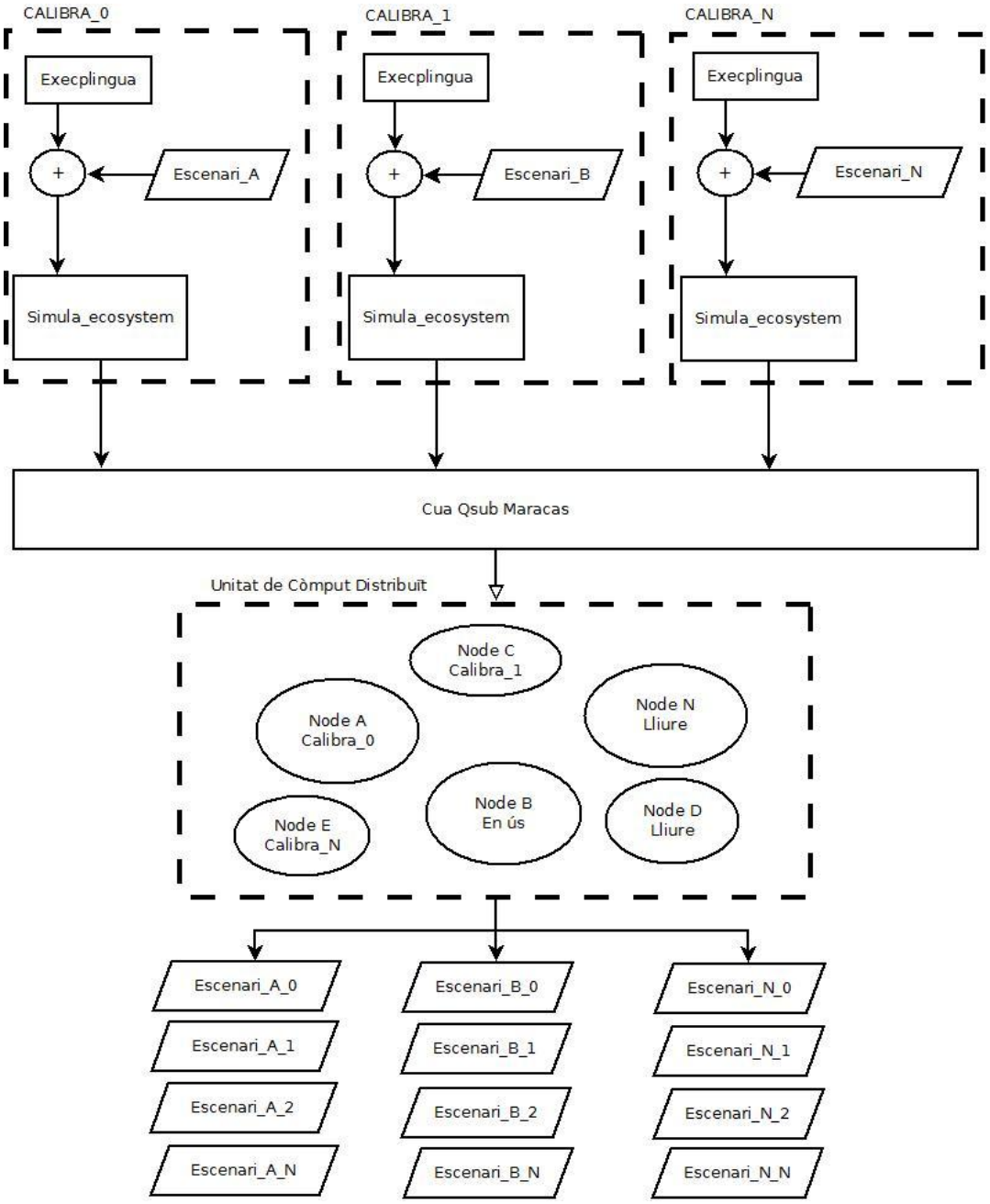

Figura 9: Procés d'obtenció de les simulacions

Inicialment l'expert ha de decidir quans ecosistemes vol processar. Quan ja ha decidit els escenaris a tractar, seran encapsulats en el seu **calibra.sh**. Internament en l'arxiu calibra, tindrem una crida a la funció **ecosystem.py**, la qual s'encarregarà de definir el nombre de simulacions que es vol generar de l'escenari, el directori on tenim les dades de l'escenari a estudiar, la ruta destí dels resultats de les simulacions generades i per últim el simulador d'ecosistemes PlinguaCore.

3.1.1 Implementació del procés de simulació d'escenaris

Un cop hem definit el procés, és el moment de presentar l'algoritme dissenyat per obtenir les simulacions d'un escenari. Gràcies a la simplificació que ens aporta l'algoritme, podem veure el procés d'implementació que s'ha realitzat per obtenir les simulacions.

Inicialment en l'Algoritme 1, presentarem els paràmetres requerits per poder executar l'script, els quals son:

- **escenari:** Variable on guardarem el directori més el nom de l'arxiu amb les dades de l'escenari que l'expert vol simular.
- **steps:** Variable on emmagatzemarem un enter,que defineix el nombre de passos a simular per cada unitat temporal: dia, mes, any. Aquest valor el defineix l'usuari expert segons el model i és un paràmetre important pel PlinguaCore.
- **N:** En aquesta variable emmagatzemarem el nombre de simulacions que l'expert vol realitzar per l'escenari.
- **directori_destí:** Variable on emmagatzemarem la ruta on guardarem les simulacions de l'escenari.

Algorithm 1 Simulació escenari (ecosystem.py)

Require: escenari, steps, N, directori_destí.

```
1: i := 0
2: while (i < N) do
3:     simula_escenari(escenari, steps, directori_destí)
4:     i := i+1
5: end while
6: return simulació_escenari
```

Un cop definits els paràmetres necessaris, l'script simularà N cop el **escenari** especificat. Per fer-ho definim un iterador (Línies 1-4) limitat pel número de simulacions **N**.

Per cada iteració, l'script crida a la funció ***simula_escenari(i)*** (Línea 3) que s'encarrega de simular l'escenari corresponent utilitzant l'script **exeplingua.sh** i emmagatzemant les dades en el directori **directori_destí**. Cal tenir present que el resultat de cada simulació s'emmagatzema en un arxiu amb el nom de l'escenari contingut en la variable **escenari**, seguit del índex de la iteració **i**, així els arxius generats s'anomenaran: **escenari_0.dat**, **escenari_1.dat**, **escenari_2.dat**, ..., **escenari_N.dat**.

3.1.2 Consideracions dels problemes apareguts en la implementació del simulador d'escenari

El problema que vam tenir a l'hora d'implementar l'script, va ser, que per cada iteració del bucle encarregat de generar les múltiples simulacions, era necessari realitzar múltiples crides sobre l'script **execplingua.sh** implementat en Bash.

La primera crida a **execplingua.sh** la realitzava sense problemes, però les posteriors, no s'executaven. Ens vam adonar, que era necessari utilitzar una llibreria que ens facilites l'execució de subprocessos per poder fer crides externes entre scripts. Python té una llibreria de funcions que ens facilita molt aquesta tasca anomenat *Subprocess management* [20]. Entre les funcions que podem trobar hi ha la ***spawnv()***, utilitzada per solucionar el problema.

La funció ***spawnv*** per poder ser executada, requereix el camps detallats a continuació:

- El primer camp, indica quin mode d'execució volem, és a dir, en el nostre cas el mode escollit és el d'espera a que finalitzi **simula_ecosystem.py**.
- El segon camp, és el directori on trobem l'script a executar
- El tercer camp ,els arguments d'execució que li passem a l'script.

3.2 Procés d'extracció d'informació

Un cop obtenim els resultats de les simulacions de cada escenari, entra en joc la implementació de la API de processament d'ecosistemes, implementada en aquests projecte. Aquesta API, s'encarregarà de processar els resultats dels diferents escenaris i de calcular l'error obtingut en les simulacions, respecte unes dades considerades ideals i proporcionades per l'usuari expert.

La primera funció de la API anomenada ***crida_parser()***, s'encarrega d'extreure dels arxius de text generats pel simulador aquella informació que és pot processar. Ja que els arxius resultants de les simulacions estan plens d'informació extra no necessària pel procés. Això es fa a través d'un script anomenat **parser.py**.

En la Figura 10 podem veure un diagrama de flux dels paràmetres d'entrada i els de sortida necessaris per executar el **parser.py**.

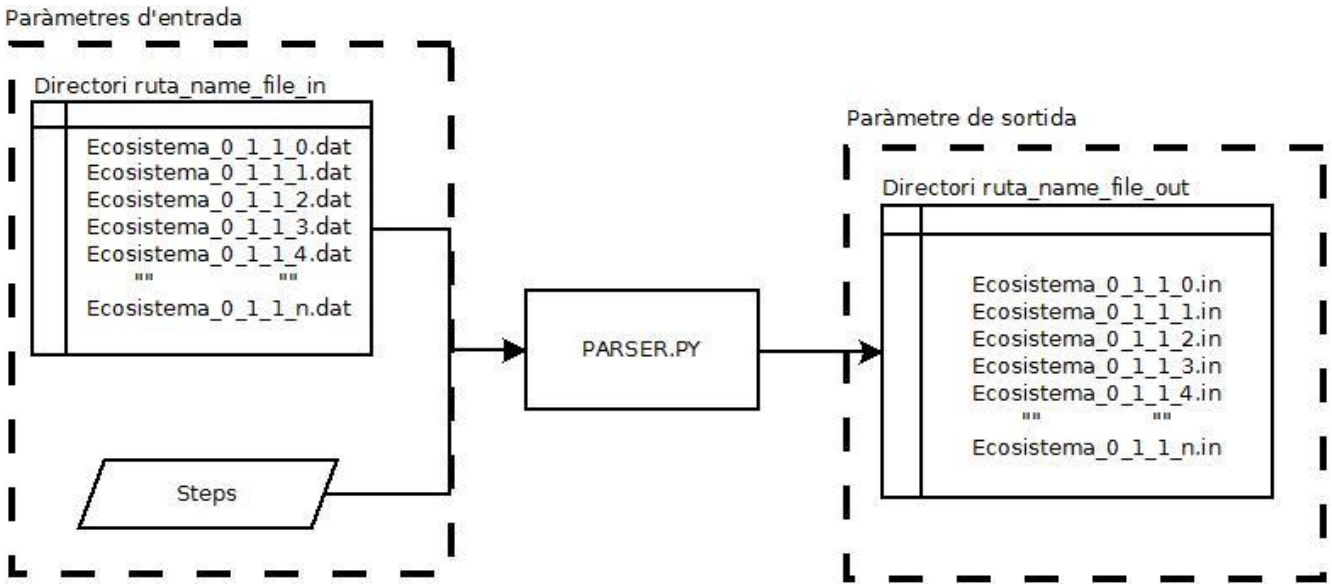

Figura 10: Paràmetres d'entrada i sortida en la funció parser

En el diagrama mostrat en la Figura 10, el primer paràmetre d'entrada és la ***ruta_name_file_in*** on hi ha emmagatzemats els arxius generats pel **simula_ecosystem.py**, indexats per diferencia'ls amb el número de la simulació, aquest valor anirà del 0 fins a N-1. El segon paràmetre, indicarà al **parser.py** en quins steps s'ha de capturar la informació, és a dir, l'step és una unitat temporal (setmanes, mesos, anys, etc.), definida per l'usuari expert, amb la qual podem obtenir la informació necessària per al model que estiguem utilitzant, que en el cas del model del Tritó Pririnenc [13] pren un valor de 17. Un cop el nostre script rep aquest valor, parsejarà la informació de la simulació cada 17 steps. Com a paràmetre de sortida requerirà que li passem el directori de sortida dels fitxers parsejats, on s'emmagatzemarà les simulacions modificant l'extensió dels arxius, de ***.dat*** a ***.in***. La variable utilitzada per emmagatzemar la ruta és ***ruta_name_file_out***.

De l'arxiu on trobem la informació de l'ecosistema Figura 11, ens interessarà captura la informació següent: el camp **Label** que ens indica quina membrana estem tractant, el camp **Multiset** on trobem els individus de l'ecosistema i el camp **Configuration** que ens servirà per identificar quin any d'estudi estem tractant.

newt01_0_0_0.dat

```
4  CONFIGURATION: 0
5  TIME: 0.0 s.
6  MEMORY USED: 216384
7  FREE MEMORY: 139274
8  TOTAL MEMORY: 1800448
9
10 MEMBRANE ID: 4, Label: 11,111, Charge: 0
11 Multiset: #
12 Parent membrane ID: 3
13
14 MEMBRANE ID: 6, Label: 21,111, Charge: 0
15 Multiset: #
16 Parent membrane ID: 5
17
18 MEMBRANE ID: 8, Label: 31,111, Charge: 0
19 Multiset: #
20 Parent membrane ID: 7
21
22 MEMBRANE ID: 10, Label: 41,111, Charge: 0
23 Multiset: #
24 Parent membrane ID: 9
25
26 MEMBRANE ID: 12, Label: 51,111, Charge: 0
27 Multiset: #
28 Parent membrane ID: 11
29
30 MEMBRANE ID: 14, Label: 61,111, Charge: 0
31 Multiset: #
32 Parent membrane ID: 13
33
```

Figura 11: Arxiu amb la informació a parsejar

3.2.1 Implementació del procés d'extracció d'informació

Un cop hem definit el procés d'extracció d'informació, és el moment de presentar l'algoritme que hem dissenyat per poder explicar el procés de parseig. Aquest procés, es repetirà per totes les simulacions sigui quin sigui l'escenari a processar.

Inicialment serà necessari realitzar un parseig previ, realitzat per l'Algoritme 2, el qual extraurà la informació estrictament necessària de cada simulació, capturarem solament la informació que es troba en les configuracions múltiples de la variable **step** i descartant la informació restant.

Per poder executar l'Algoritme 2 necessitarem el directori més el nom de la simulació a tractar, **dat_file** i també necessitarem la variable **steps** que ens indicarà cada quan capturem la informació.

Algorithm 2 Extracció d'informació (Llista amb la informació per configuració)

```
Require: steps, dat_file.
 1: file_escenari := llegir_escenari(dat_file)
 2: for line in file_escenari do
 3:    tag := line.split(": ")
 4:    if (tag[0] == "CONFIGURACIO") then
 5:       num_confi := confi[1]
 6:       if num_config%steps == 0 then
 7:          lst_item_configuracio := capturem_item_configuracio()
 8:          lst_configuracions.Append(lst_item_configuracio)
 9:       end if
10:    end if
11: end for
```

En la (Línea 1) llegirem l'escenari que trobem en **dat_file** i el guardem sobre la variable **file_escenari**. Posteriorment, crearem un bucle ***for*** que ens servirà per llegir l'escenari que hem capturat, i d'aquesta forma podrem anar filtrant la informació que ens interessa (Línia 2-10). Per obtenir-la, serà necessari realitzar els següents passos:

1. Crearem un objecte que li direm **tag**, aquest ens servirà per poder capturar la informació que ens interessa de l'arxiu (Línia 3).
2. Capturarem totes les configuracions que trobem en l'arxiu i les guardarem sobre una llista que li direm **num_config** (Línies 4-9).
3. Per cada configuració **num_config**, mirarem si és múltiple de la variable steps (Línies 6-9), si es així :
    - Capturarem tota la informació que trobem en la configuració que estem tractant i l'emmagatzemarem en la llista **lst_item_configuracio** (Línia 7)**.**
    - Un cop tenim tota la informació de la configuració en **lst_item_configuracio**, guardarem aquesta informació a la primera posició buida de **lst_configurations** (Línia 8).

Un cop tenim la informació que hi ha dintre de les configuracions, l'objectiu és retornar per cada simulació, un arxiu on aparegui ordenat per membranes: l'any d'estudi, el número de la configuració (que serà igual al quocient entre **configuration** i **steps)**, el camp **label** que ens indicarà la càrrega de la membrana i finalment el camp **multiset** on trobem els individus a estudiar.

l'Algoritme 3 mostrat a continuació ens retornarà un arxiu per simulació. Inicialment definirem els paràmetres d'entrada necessaris per poder executar l'algoritme, els quals son: la variable **steps**, **directori_in** on guardem cada arxiu parsejat i **lst_configuracions**.

Algorithm 3 Extracció d'informació (Obtenim les dades de la simulació)

```
Require: steps, directori_in, lst_configuracions
 1: for configuracio in lst_configuracions do
 2:   for items in configuracio do
 3:     tag_item := capturem_tag(items)
 4:     if (tag_item == "CONFIGURATIO") then
 5:       id_configuration := capturem_id(items)
 6:     else if (tag_item == "MEMBRANE ID") then
 7:       lebel_membrane := capturem_lebel(items)
 8:     else if (tag_item == "Multiset") then
 9:       dades_especie := capturem_dades(items)
10:       dades := id_configuration + lebel_membrane + dades_especie
11:       escriure(directori_in, dades)
12:     end if
13:   end for
14: end for
```

Per poder fer el parseig crearem un bucle ***for*** que anirà recorren la llista **lst_configuracions** (llista amb els pasos de simulació múltiples de la variable step), així podrem anar fent un filtratge de les dades que trobem en l'arxiu(Línia 1-14).

- Crearem un objecte que li direm **tag_item**, aquest objecte ens servirà per poder capturar la informació que ens interessa (Línia 3).
- Posteriorment mirarem si **tag_item** és igual a ***CONFIGURATION***, si és així capturarem el camp **configuration** i guardarem en la variable **id_configuration** el número de la configuració que estem tractant (Línies 4-6).
- Seguidament mirarem si **tag_item** és igual a ***MEMBRANE ID***, si és així capturem el camp **lebel** que ens servirà per identificar la membrana que estem tractant, el valor de la membrana el guardarem sobre **lebel_membrane** (Línia 6-8) .
- Seguidament mirarem si el **tag_item** a trobat el cap ***Multiset***, si és així capturarem les dades que trobem en el **Multiset** i les guardarem sobre **dades_especie** (Línia 8-12). Per finalitzar el procés concatenarem els camps capturats (**id_configuration + lebel_membrane + dades_especie**) i guardarem el resultat sobre **dades** (Línea 10).

Ara solament quedarà escriure les dades emmagatzemades de l'string **dades** a un nou fitxer amb extensió ***.out***. Per poder realitzar aquest procés, necessitarem l'string **dades** i el paràmetre **directori_in** on guardem la ruta més el nom de la simulació processada (Línia 11). El procés l'anirem repetint per totes les simulacions de l'escenari.

En la Figura 12 podem veure l'arxiu que obtenim per cada simulació de l'escenari estudiat un cop aplicat l'script d'extracció d'Informació

*newt01_1_1_0_0_0_0_0_0_0_0_0_0_20.in

```
 3 1:17:3,111:RI{1}, RI{2}, R{0}, X{1,12}, X{1,3}*727, X{1,11}*3, X{1,2}*85, X{1,9}*34, X{1,7}*72, X{1,6}*19, X{1,4}*390, X
   {1,22}, X{1,1}*58, X{1,13}*6, X{1,19}, X{1,5}*166, X{1,17}*2, X{1,15}*6, X{1,8}*27, X{1,10}*3
 4 1:17:4,111:RI{1}, RI{2}, R{0}, X{1,11}*4, X{1,5}*172, X{1,2}*77, X{1,1}*58, X{1,6}*19, X{1,4}*395, X{1,15}*4, X{1,10}*2, X
   {1,9}*20, X{1,8}*31, X{1,7}*72, X{1,3}*743, X{1,17}*2, X{1,13}*6
 5 1:17:5,111:RI{1}, RI{2}, R{0}, X{1,10}*6, X{1,8}*67, X{1,23}, X{1,25}, X{1,17}*5, X{1,2}*142, X{1,5}*348, X{1,4}*757, X
   {1,13}*10, X{1,12}, X{1,6}*41, X{1,26}, X{1,3}*1459, X{1,1}*132, X{1,11}*5, X{1,7}*127, X{1,15}*4, X{1,9}*48, X{1,22}*2
 6 1:17:6,111:RI{1}, RI{2}, R{0}, X{1,15}*12, X{1,5}*508, X{1,1}*168, X{1,2}*233, X{1,3}*2171, X{1,8}*84, X{1,17}*5, X{1,13}
   *16, X{1,7}*196, X{1,14}, X{1,9}*38, X{1,6}*64, X{1,21}*3, X{1,23}, X{1,10}*7, X{1,19}*2, X{1,18}*2, X{1,11}*12, X{1,4}
   *1152
 7 1:17:7,111:RI{1}, RI{2}, R{0}, X{1,13}*10, X{1,8}*29, X{1,4}*384, X{1,17}*2, X{1,2}*69, X{1,10}*3, X{1,21}*2, X{1,22}, X
   {1,14}, X{1,12}, X{1,7}*69, X{1,1}*50, X{1,6}*19, X{1,11}*2, X{1,5}*165, X{1,15}*6, X{1,9}*25, X{1,3}*722
 8 1:17:8,111:RI{1}, RI{2}, R{0}, X{1,2}*69, X{1,10}*4, X{1,13}*5, X{1,15}*5, X{1,11}*4, X{1,18}*2, X{1,21}, X{1,9}*19, X{1,4}
   *406, X{1,6}*23, X{1,5}*166, X{1,7}*73, X{1,14}, X{1,3}*738, X{1,8}*33, X{1,1}*67
 9 1:17:9,111:RI{1}, RI{2}, R{0}, X{1,5}*177, X{1,9}*24, X{1,17}*3, X{1,11}*2, X{1,1}*71, X{1,8}*34, X{1,3}*713, X{1,22}, X
   {1,18}*2, X{1,21}, X{1,2}*75, X{1,13}*5, X{1,4}*367, X{1,6}*29, X{1,12}, X{1,19}, X{1,7}*71
10 1:17:0,111:F{2,6}, F{2,4}, F{2,3}, F{2,9}, F{2,5}, F{2,2}, F{2,7}, F{2,8}, F{2,1}
11 1:17:111,111:#
12 1:17:p:#
13 2:34:1,111:RI{1}, RI{2}, R{0}, X{1,23}, X{1,4}*2814, X{1,14}*19, X{1,22}*5, X{1,7}*63, X{1,8}*272, X{1,2}*220, X{1,9}*115,
   X{1,16}*17, X{1,20}*3, X{1,3}*265, X{1,11}*12, X{1,19}*3, X{1,5}*1456, X{1,13}*4, X{1,6}*653, X{1,1}*5857, X{1,12}*10, X
   {1,18}*7, X{1,15}*2, X{1,10}*71
14 2:34:2,111:RI{1}, RI{2}, R{0}, X{1,16}*2, X{1,4}*1240, X{1,5}*647, X{1,7}*39, X{1,19}*3, X{1,12}*7, X{1,20}*2, X{1,14}*12,
   X{1,2}*72, X{1,9}*48, X{1,10}*44, X{1,6}*315, X{1,8}*143, X{2,1}*231, X{1,23}*2, X{2,5}*197, X{1,11}*8, X{1,3}*121, X
   {1,15}, X{1,17}, X{1,24}, X{1,1}*1370, X{2,4}*370, X{1,18}*5
15 2:34:3,111:RI{1}, RI{2}, R{0}, X{1,2}*55, X{1,1}*1480, X{1,4}*679, X{1,13}, X{1,20}, X{1,16}*6, X{1,12}*3, X{1,18}*2, X
   {1,8}*70, X{1,7}*17, X{1,11}*3, X{1,10}*34, X{1,5}*371, X{1,14}*6, X{1,6}*157, X{1,3}*82, X{1,23}, X{1,9}*41
16 2:34:4,111:RI{1}, RI{2}, R{0}, X{1,4}*707, X{1,14}*5, X{1,2}*56, X{1,11}*2, X{1,12}*4, X{1,18}*2, X{1,8}*70, X{1,5}*376, X
   {1,9}*28, X{1,7}*17, X{1,16}*3, X{1,1}*1309, X{1,10}*19, X{1,3}*76, X{1,6}*163
17 2:34:5,111:RI{1}, RI{2}, R{0}, X{1,26}, X{1,24}, X{1,16}*4, X{1,4}*1390, X{1,12}*5, X{1,18}*5, X{1,13}, X{1,9}*67, X{1,2}
   *126, X{1,10}*47, X{1,7}*38, X{1,14}*10, X{1,23}*2, X{1,5}*717, X{1,3}*138, X{1,1}*2833, X{1,11}*6, X{1,8}*123, X{1,6}*331
18 2:34:6,111:RI{1}, RI{2}, R{0}, X{1,10}*37, X{1,14}*15, X{1,9}*61, X{1,7}*58, X{1,2}*164, X{1,6}*479, X{1,19}*2, X{1,24}, X
```

Figura 12: Arxiu obtingut d'aplicar el parser.py

3.2.2 Consideracions dels problemes apareguts en el procés d'extracció d'informació

En aquest primer script de l'API és interessant destacar les estructures utilitzades per l'execució del mateix. Aquestes estructures han estat les ***llistes*** com a contenidors de text i els ***strings*** per poder parsejar la informació redundant retornada pel simulador.

També cal destacar, ja que ha estat el primer script implementat per nosaltres, ens a suposat un temps extra en la seva implementació. Ja que ens hem tingut que documentar per poder programar en Python.

3.3 Procés de filtratge per espècies

Un cop realitzat el primer parseig a mans del **parser.py**, ens trobem que en el directori ***file_in***, s'han generat tants arxius amb extensió ***.in*** com simulacions de l'ecosistema a parsejat el **parser.py**. La tasca que realitzarà la funció ***cirda_compiler()***, és unir la informació de l'espècie a estudiar, continguda en cada membrada, és a dir, totes les membranes que corresponen al mateix any, com podem veure en la Figura 12 mostrada en el punt (3.2.1) les processen per un cantó, seguidament les del següent any i així fins completar el tractament per tots els anys.

Per realitzar aquest procés, utilitzarem una estratègia de ***Hash*** o el que és el mateix en Python, un ***Diccionari***. Tot diccionari esta format per claus i valor, en el nostre cas, les claus del diccionari seran les espècies que tenim en l'arxiu editable.

Per poder decidir amb quina o quines espècies vol treballar l'expert, hem dissenyat un arxiu editable que prèviament l'execució de l'script, serà editat per l'usuari expert, amb el nom de l'espècie o les espècies a estudiar. En la Figura 13, podem veure l'arxiu editable, amb un exemple d'espècies a estudiar.

Figura 13: Arxiu editable amb espècies.

Finalment a l'acabar el parseig, en el directori destí ***file_out*** generarem un arxiu de sortida amb extensió ***.out***, el qual tindrà les dades de l'ecosistema amb un resum de les espècies definides per l'usuari expert.

El diagrama de flux que veiem en la Figura 14, ens mostra els paràmetres d'entrada i els de sortida necessaris per poder executar l'script ***compiler.py***.

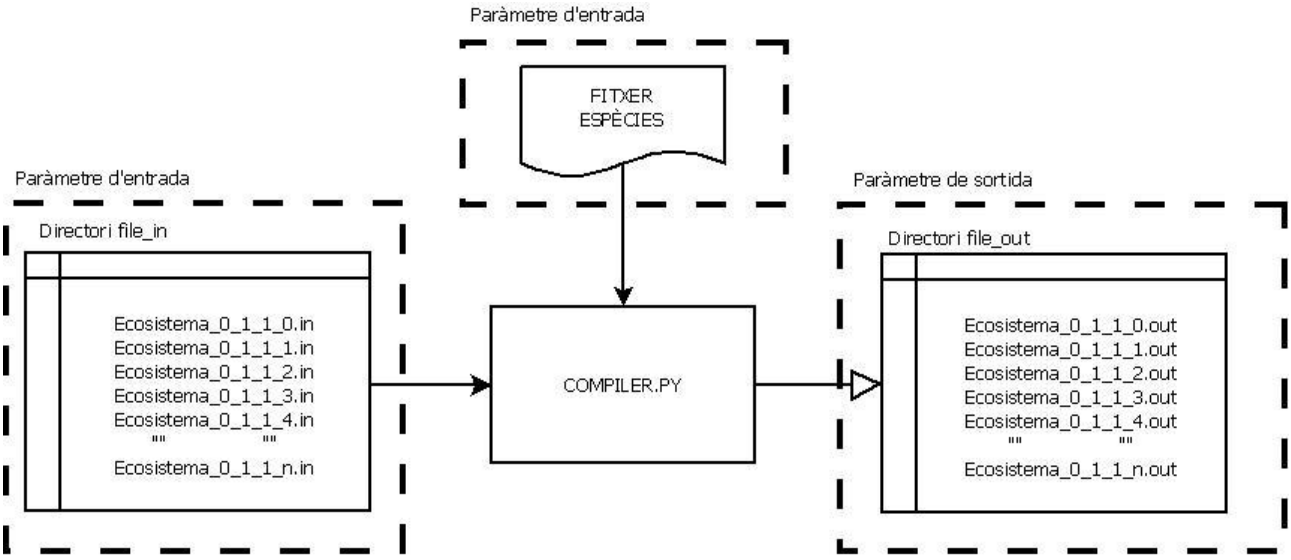

Figura 14: Paràmetres d'entrada i de sortida de l'script compiler.py

3.3.1 Implementació del procés de filtratge per espècies

Un cop hem definit el procés de filtratge per espècie en el punt anterior, és el moment de presentar l'algoritme que hem dissenyat per poder explicar detalladament la implementació que hem realitzat.

l'Algoritme 4, s'encarregarà de filtrar els arxius, tenint en compte de quin escenari son simulació. Els paràmetres requerits per poder executar l'algoritme seran els següents: **directori_in** on emmagatzemem el directori més el nom de l'escenari a tractar, **directori_out** on escriurem els arxius processats per la funció ***crida_compiler()***, la ruta a l'arxiu editable emmagatzemat en **arxiu_especies** i finalment el nom de l'escenari emmagatzemat en la variable **scenaryname**.

Algorithm 4 Filtratge per espècies (obtenim els escenaris a tractar)

```
Require: directori_in, directori_out, arxiu_especies, scenaryname.
1: lst_arxius := filtrar_arxius(scenaryname, directori_in)
2: for (i:=0 ; i<len(lst_arxius) ; i++) do
3:    processar_arxiu(lst_arxius(i), directori_in, arxiu_especies, scenary-
      name)
4: end for
```

Els passos que realitzarà l'algoritme seran els detallats a continuació:

1. Amb la variable **scenary_name** i el **directori_in** executarem la funció ***filtrar_arxiu()***, aquesta funció capturarà en una llista que li direm **lst_arxius**, tots els arxius que pertanyen a l'escenari que estem estudiant (Línea 1).
2. Crearem un bucle ***for*** que ens recorri la llista **lst_arxius** (Línes 2-4), i per cada arxiu executarem la funció ***processa_arxiu()*** (Línia 3), amb els paràmetres: **lst_arxius[i]**, **directori_out**, **directori_in**, **arxiu_especies** i **scenaryname**. Encarregada de fer el procés de filtratge per cada arxiu de la simulació.

Posteriorment, en l'Algoritme 5 podrem veure detalladament la funció ***processa_arxiu()***. Els paràmetres requerits seran els següents: **lst_arxius, directori_out, arxiu_especies** i **scenaryname**.

Algorithm 5 Filtratge per espècies

```
1: inicialitzar_diccionari(dic)
2: file := open(scenaryname, directori_in)
3: any_actual := 1
4: for line in file do
5:    linia := line.split(":")
6:    any := obtenir_any(linia)
7:    if (any <> anyactual) then
8:       escriure_resultats(dic, directori_out, nom_arxiu, any_actual)
9:       inicialitzar_diccionari(dic)
10:   end if
11:   filtrar_especies(linia, arxiu_especies, dic)
12: end for
```

Per començat, inicialitzarem el diccionari i com a claus les espècies que l'usuari expert vol processar i que ha definit en l'arxiu dins la variable **arxiu_especies**. Cada clau s'inicialitza amb valor zero (Línia 1). Posteriorment, obrirem la simulació a analitzar i la guardarem sobre l'objecte **file** (Línea 2). Per tenir controlat el nombre de simulacions processades crearem un comptador que li direm **any_actual** i l'iniciarem a 1 (Línia 3). Per poder realitzar el procés de filtratge serà necessari crear un bucle ***for*** que recorri la simulació que tenim en l'objecte **file** (Línia 4-12), per cada línea de la simulació, aplicarem el següent filtratge:

1. Inicialment crearem l'objecte **linia,** aquest objecte ens servirà per poder capturar la informació que ens interessi per cada línea de l'arxiu (Línea 5).
2. Capturarem el primer any utilitzant la funció ***obtenir_any(linia)***, i el guardarem sobre **any** (Línia 6).
3. Comprovarem si la variable **any** es diferent al **any_actual** (Línia 7-10), si no és així, significarà que la membrana que estem tractant pertany a l'any d'estudi i serà necessari filtrar-la utilitzant la funció ***filtrar_especies*** (Línia 11). En el cas contrari significa que hem filtrat totes les membranes que formen l'any d'estudi i serà necessari escriure els valors filtrats en l'arxiu, utilitzant la funció ***escriure_resultats*** (Línia 8). Per acabar tornem a inicialitzar el diccionari (Línia 9).

En la Figura 15, podem veure els resultats obtinguts després de realitzar l'estudi per l'espècie X{1}, durant 19 anys.

newt01_1_1_0_0_0_0_0_0_0_0_0_0_0_0.out

```
1:X{1}*24967
2:X{1}*44824
3:X{1}*80984
4:X{1}*122885
5:X{1}*156524
6:X{1}*125368
7:X{1}*171768
8:X{1}*153680
9:X{1}*204038
10:X{1}*224277
11:X{1}*188551
12:X{1}*222194
13:X{1}*153439
14:X{1}*219434
15:X{1}*153306
16:X{1}*219598
17:X{1}*226395
18:X{1}*230287
19:X{1}*231648
```

Figura 15: Arxiu després de l'execució del compiler.py

3.3.2 Consideracions dels problemes apareguts en el procés de filtratge per espècies

En la implementació del **compiler.py** van sorgir una sèrie de dificultats a l'hora de realitzar la implementació. La primera dificultat va ser la necessitat d'utilitzar una estratègia de ***Hash*** per tal de guardar els valors de cada membrana per cada any d'estudi, i la següent dificultat va ser la utilització de patrons per tal d'identificar objectes a l'interior d'una cadena.

Inicialment per resoldre el primer problema, es va pensar amb crear una matriu, la qual tingues unes dimensions fixades per el nombre d'espècies a analitzar i els anys a analitzar. Però ràpidament es va descartar la opció perquè requeria guardar en memòria molta informació i vam pensar que el més òptim seria crear un diccionari el qual ens permet guardar una clau que serà l'espècie a analitzar i el seu valor, el qual seria modificat sumant per aquesta clau el nombre d'individus que trobem.

La següent dificultat que va sorgir, es la utilització de patrons. Estava clar, que era necessari comparar entre les espècies que tinc en l'arxiu editable i les que tinc en les membranes a estudiar. Un cop tinc les espècies de cada membranes a estudiar a l'interior de una llista, sol calia comparar si coincidia l'espècie de la llista amb les espècies a estudiar per l'expert en l'arxiu editable, en el nostre cas el patró. D'aquesta forma podria tractar totes les espècies d'una forma ràpida i eficient.

3.4 Procés d'obtenció dels valors estadístics

Un cop finalitzat el parseig que ha realitzat el **compiler.py**, obtindrem de cada simulació un arxiu, que contindrà l'espècie o les espècies a estudiar amb el nombre d'individus per cada any que volem tractar. Serà en el ***Directori_out***, on emmagatzemarem els resultats d'executar el **compiler.py**. Un cop tenim aquesta informació, ja podem obtenir resultats interessants per l'usuari expert, com pot ser la mitjana de totes les simulacions o la desviació.

Per obtenir aquest resultats, hem implementat la funció ***crida_process()***, que farà una crida a l'script **process.py**, i tractarà els arxius emmagatzemats en el directori ***file_out***. Els resultats tant de la desviació, com la mitjana per cada any, seran emmagatzemats en el directori ***Estadístics***, i el nom dels arxius resultants serà el nom de l'escenari que hem simulat, amb extensió ***.avg*** si estem parlant de mitjana o ***.dev*** si és el cas de la desviació.

En el diagrama de flux mostrat en la Figura 16, podem veure els paràmetre d'entrada i sortida, utilitzats per l'script **process.py**. Els paràmetres d'entrada seran; el nom de l'escenari a processar. A continuació, l'arxiu d'espècies, el qual ens servirà per identificar les espècies a analitzar i el directori on trobem els resultats obtinguts un cop finalitzat el **compiler.py**. Mentre que el paràmetre de sortida es el directori **Estadístics**, on s'emmagatzema els resultats de la mitjana i la desviació per escenari i escenari.

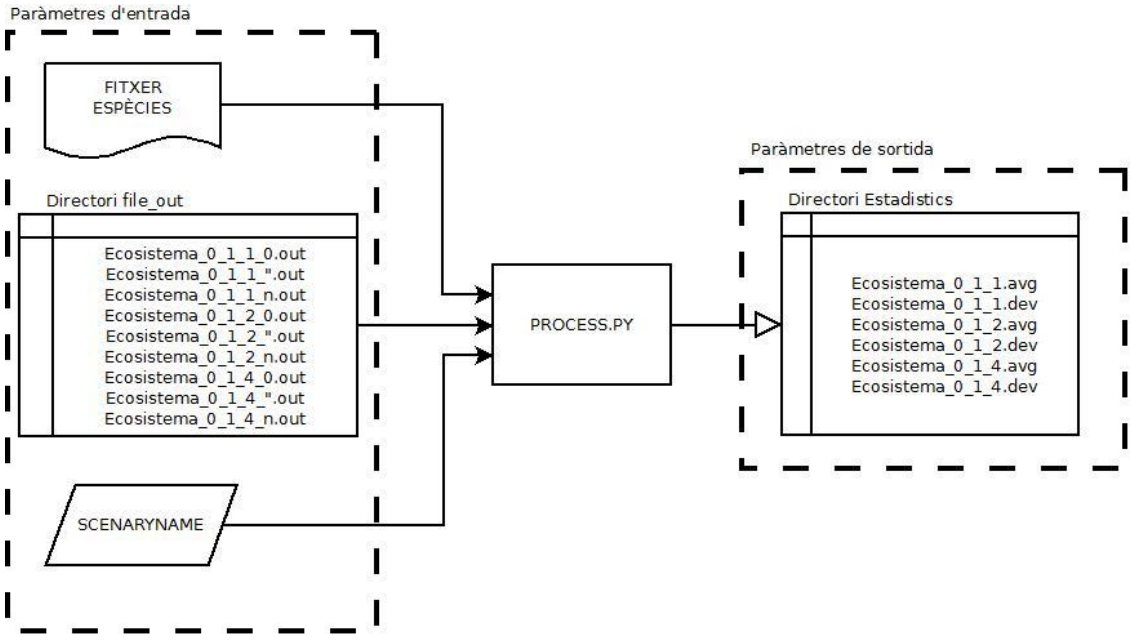

Figura 16: Diagrama dels paràmetres requerits per executar el process

3.4.1 Implementació del sistema d'obtenció dels valors estadístics

En l'algorisme 6, és presenta l'algorisme necessari per obtenir el resum estadístic de la informació sol·licitada per l'usuari expert. Concretament en aquest treball ens hem centrat en l'obtenció dels valors promig i desviacions per any, de les especies sol·licitades per l'usuari expert, obtingudes a partir de totes les simulacions d'un mateix escenari. Per poder executar l'algoritme, necessitarem els següents paràmetres:

- **Scenaryname:** Variable on hi ha el nom de l'escenari que estem tractant.
- **Pathespecies:** Contindrà el directori més el nom del fitxer d'espècies.
- **Pathestadistics:** Directori on guardarem els resultats derivats de l'execució de la funció ***crida_process()***. El directori on emmagatzema la informació parsejada és el **estadistics**.
- **Directori_out:** Aquesta variable contindrà el directori on trobem les simulacions generades per la funció ***crida_compiler()***. Així les podrem tractar i obtenir els valors estadístics.

Algorithm 6 Càlcul estadístic de les simulacions

Require: Scenaryname, Directori_out, Pathespecies, Pathestadistics.

```
capturem_file := llegir_fitxer_editable(Pathespecies)
lst_especies := capturar_especies(capturem_file)
lst_arxius := capturar_simulacio(Scenaryname, Directori_out)
for arxiu in lst_arxius do
    file := obrir_arxiu(arxiu)
    for individu in file do
        lst_individus := add_individus(individu)
    end for
    lst_acumuladora_individus := Append(lst_individus)
end for
resultats_mitjana := promitg(lst_acumuladora)
resultats_desviacio := desviacio(lst_acumuladora)
for items in resultats_mitjana do
    escriure_mitjana(items, Scenaryname)
end for
for items_2 in resultats_desviacio do
    escriure_desviacio(items_2, Scenaryname)
end for
```

Inicialment capturarem l'arxiu editable on l'expert introdueix les espècies que vol tractar (Línia 1), i les guardem en la llista **lst_especies** (Línia 2). Posteriorment, guardarem sobre **lst_arxius** tots els arxius que coincideixen amb **scenaryname** i es trobin en **Directori_out** (Línia 3).

Per poder acumular tots els individus que tenim per simulació de l'escenari, serà necessari implementar un bucle ***for*** (Línia 4-10). Aquest bucle, per cada iteració llegirà un fitxer que el trobarem emmagatzemat a la llista **lst_arxius** i aplicarem el següent filtratge:

- Lectura del primer arxiu de la llista capturat per l'objecte **file** (Línia 5).
- Crearem un bucle ***for*** (Línia 6-8) que anirà acumulant els individus que troba en la simulació, d'aquesta forma en la llista **lst_individus** (Línia 7) tindrem en cada posició de la llista, el nombre d'individus i l'any d'anàlisis en que s'ha trobat els individus.
- Quan ja tenim acumulats tots els individus de la simulació, emmagatzemarem la llista **lst_individus** en una nova llista acumuladora per tal de tenir en cada posició de la llista acumuladora els individus extrets de cada simulació (Línia 9).

El procés s'anirà repetint fins arribar a l'última posició de la llista **lst_arxius** (Línia 3). Al finalitzar el procés, en cada posició de la llista acumuladora tindrem les dades que necessita l'expert per fer els càlculs estadístics. Seguidament, farem el promig (Línia 11) i la desviació (Línia 12) de les dades que tenim tant en **resultats_mitjana** com **resultats_desviacio**. Ara sol ens quedarà escriure els resultats obtinguts en un nou arxiu amb extensió ***.avg*** per la mitjana i amb extensió ***.dev*** per la desviació.

En la Figures 17 i la Figura 18 podem veure els arxiu resultants del filtratge que a realitzat la funció ***crida_process()***.

newt01_0_1.avg

```
1:X{1}*28560.18
2:X{1}*37954.8
3:X{1}*58876.54
4:X{1}*93702.32
5:X{1}*125686.3
6:X{1}*85461.48
7:X{1}*121681.66
8:X{1}*87932.1
9:X{1}*130676.22
10:X{1}*169351.88
11:X{1}*115993.6
12:X{1}*164058.5
13:X{1}*121158.04
14:X{1}*167179.88
15:X{1}*121428.62
16:X{1}*174066.48
17:X{1}*203807.58
18:X{1}*205482.64
19:X{1}*206481.6
```

Figura 17: Arxiu amb les dades de la mitjana per any simulat

newt01_0_0.dev

```
1:X{1}*66.2721661031
2:X{1}*169.63162703
3:X{1}*437.692687725
4:X{1}*640.581506654
5:X{1}*1061.89777704
6:X{1}*988.198473093
7:X{1}*1090.8977954
8:X{1}*888.490230047
9:X{1}*958.461614602
10:X{1}*2169.92201601
11:X{1}*4630.39055228
12:X{1}*6700.26511913
13:X{1}*3453.43513363
14:X{1}*6347.4363854
15:X{1}*3838.84982775
16:X{1}*6529.14089976
17:X{1}*6300.09656011
18:X{1}*4169.08071402
19:X{1}*1225.37531665
```

Figura 18: Arxiu amb les dades de la desviació per any simulat

3.4.2 Consideracions dels problemes apareguts en el procés d'obtenció dels valors estadístics

La principal problemàtica a l'hora d'implementar l'script **process.py**, ens va aparèixer a l'hora de decidir com realitzar la suma dels individus de totes les simulacions estudiades per l'expert, ja que teníem tota la informació emmagatzemada en una llista, i l'ús de matrius com a estratègia d'emmagatzemament, no és optima, ja que consumeix molts recursos de memòria principal.

Per poder sumar tots els individus de la simulació discernint per any de simulació, ha estat necessari emmagatzemar tots els individus de les simulacions en una llista de llistes anomenada **lst_acumuladora_individus**. Buscant en l'extensa llibreria de funcions de Python, vam trobar una funció que ens serviria per aparellar els individus, es a dir, els individus que fan referència al primer any de simulació, els emmagatzemaríem en una tupla si estem estudiant una única espècie per any de simulació, o varies tuples si estem treballant amb més espècies.

Gràcies aquest procés, un cop hem aparellat els individus a estudiar, tant sols a fet falta sumar els valors de la tupla, sense tenir la necessitar d'utilitzar matrius que elevaria el cost en memòria del procés.

3.5 Procés per obtenir l'escenari ideal

Un cop hem obtingut tant la mitjana aritmètica com la desviació amb l'execució del **process.py** per a tants escenaris d'un ecosistema com ha decidit estudiar l'usuari expert, és ara quan hem de decidir quina d'aquestes configuracions s'apropa més a la configuració ideal o a la que l'usuari expert considera que és la més correcta.

Per poder decidir quina és la configuració ideal, tindrem que comparar els resultats obtinguts de calcular la mitjana de les simulacions, amb els valors reals de l'ecosistema, presos en el camp per l'usuari expert i emmagatzemats en un arxiu anomenat **Ideal**. Per realitzar aquest procés, farem una crida sobre la funció ***crida_ideal()*** que s'encarregarà de fer la comparació entre les dades reals i les simulades.

Quan ja tenim comparades totes les configuracions amb l'ideal, obtindrem un arxiu amb el nom de la configuració de l'escenari que estem observant i el valor de la diferència de quadrats respecte l'escenari considerat ideal, el valor més petit obtingut del càlcul de la diferència de quadrats serà l'escenari que prendrà l'expert com a correcte.

En la Figura 19, podem veure els resultats obtinguts, en realitzar el procés per a 5 escenaris. Un cop calculada la diferència de quadrats entre els valors observats per l'expert i els simulats per la nostra API de càlcul d'ecosistemes, podem dir que l'escenari "newt01_9_0_0_0_0_0_0_0_0_0" el qual agafa el valor més pròxim a zero, es l'escenari que més s'apropa a les característiques de l'escenari preses per l'expert en el cap.

```
[mtrigueros@maracas TRITO]$ ./calibra_ideal.sh
newt01_1_1_0_0_0_0_0_0_0_0_0_0_0 : 1247853.33634
newt01_3_0_0_0_0_0_0_0_0_0_0_0_0 : 1227336.24973
newt01_4_1_0_0_0_0_0_0_0_0_0_0_0 : 1193420.74649
newt01_9_1_0_0_0_0_0_0_0_0_0_0_0 : 40.9503982477
[mtrigueros@maracas TRITO]$
```

Figura 19: Captura dels resultats dels escenaris

En el diagrama de flux mostrat en la Figura 20, podem veure els paràmetres necessaris per poder trobar la simulació de l'ecosistema que més s'apropa a l'estudiat en el medi per l'expert. Per poder executar l'script requerirem com a paràmetres d'entrada, el fitxer on l'expert té les dades de l'ecosistema observades empíricament, el directori on trobem els valors calculats en cada simulació i el nom de l'escenari que volem tractar. Un cop li passem els paràmetres a l'script **Ideal.py**, ell ens retornarà un valor decimal, aquest valor li servirà a l'expert per decidir si l'ecosistema simulat té unes característiques similars al estudiat en el medi o no.

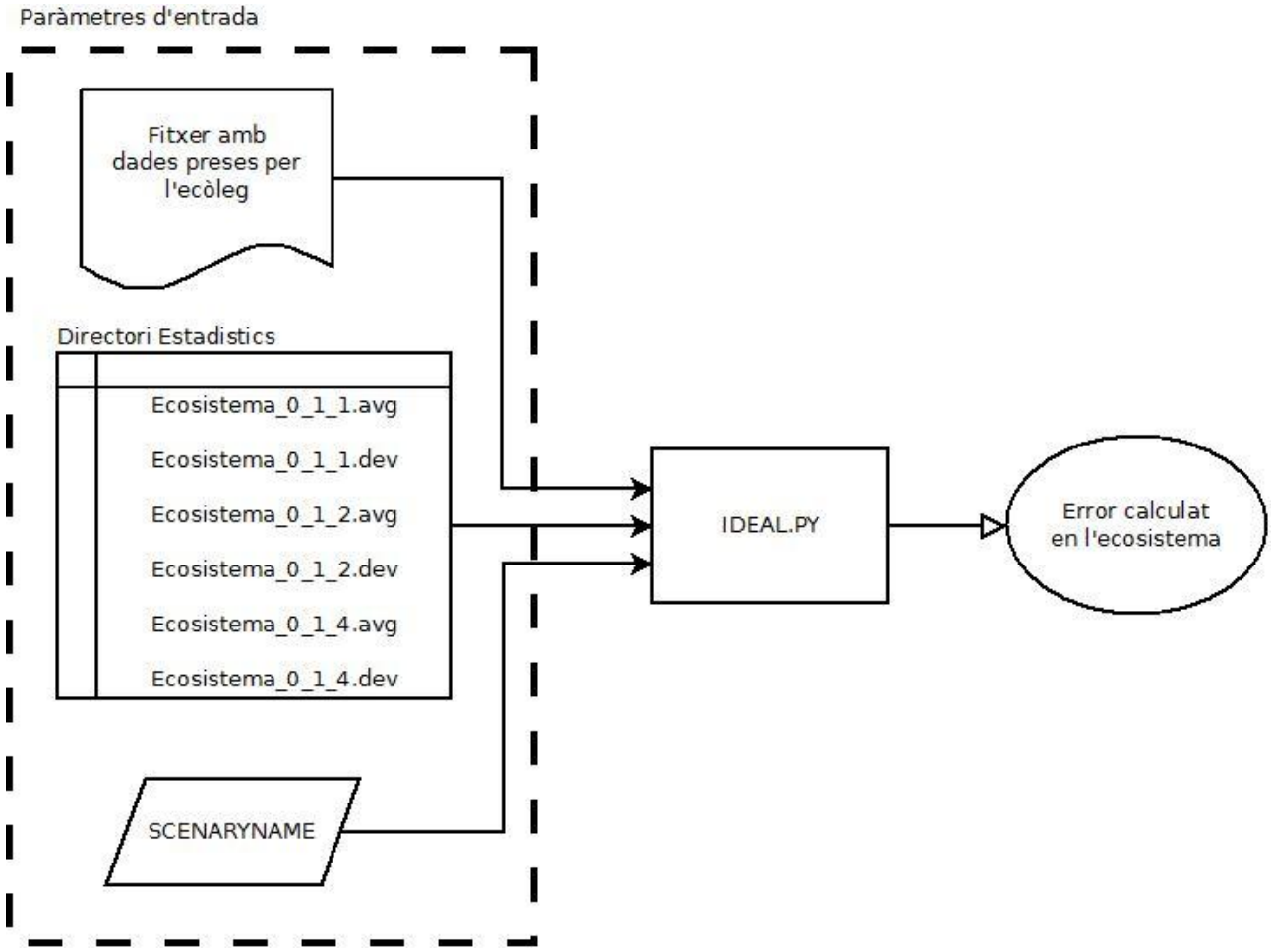

Figura 20: Paràmetres requerits per executar l'script Ideal.py

3.5.1 Implementació per obtenir l'escenari ideal

Un cop hem definit el procés per obtenir l'escenari ideal en el punt anterior, és el moment de presentar l'algoritme que hem dissenyat per poder explicar detalladament la implementació que hem realitzat.

En el Algoritme 6 podrem veure amb detall, com s'ha implementat la funció ***crida_indeal()***. Inicialment, serà necessari introduir els paràmetres d'entrada necessaris per poder executar l'algoritme:

- **Scenayname:** Variable on hi ha el nom de l'escenari que estem tractant.
- **Ideal_ecosistem:** Variable on emmagatzemem el directori més el nom de l'arxiu on l'expert a introduït les dades de l'ecosistema real.
- **Pathestadistics:** Directori on guardarem els resultats derivats de l'execució de la funció ***crida_process()***. El directori on emmagatzema la informació parsejada és el **estadistics**.

Algorithm 7 Càlcul de la diferència de quadrats

```
Require: Scenaryname, Ideal_ecosistem, Pathestadistics.
 1: Arxiu_mitjana := captura_mitjana(Scenaryname, Pathestadistics)
 2: Arxiu_ideal := capturar_ideal(Ideal_ecosistem)
 3: for items in Arxiu_mitjana do
 4:    individu := items.split("*")
 5:    lst_individus_simulacio := obtenim_valors(individu)
 6: end for
 7: for items in Arxiu_ideal do
 8:    individu := items.split("*")
 9:    lst_individus_ideals := obtenim_valors(individu)
10: end for
11: while (n<len(lst_individus_simulacio) AND n<len(lst_individus_ideal))
    do
12:    if lst_individus_ideal[n] != "" AND lst_individus_ideal > 0 then
13:       resta := lst_individus_simulacio[n] - lst_individus_ideal[n]
14:       dif_quadrat = ((resta)**2) / lst_individus_ideal[n]
15:       lst_dif_quadrat.Append(dif_quadrat)
16:    end if
17:    n := n+1
18:    dif_quadrat_escenari := suma_valors(lst_dif_quadrat)
19: end while
20: return dif_quadrat_escenari
```

El primer pas que cal fer, és capturar l'arxiu on tenim emmagatzemats els valors de la mitjana aritmètica per any simulat. Per realitzar aquest procés, li passarem a la funció ***captura_mitjana()*** amb els paràmetres: **Scenaryname** i **Pathestadistics**, posteriorment guardarem l'arxiu sobre l'objecte **Arxiu_mitjana** (Línia 1).

A continuació, realitzarem el mateix procés per obtenir l'arxiu on apareixen les dades extretes empíricament de l'ecosistema. Li passarem a la funció ***captura_ideal()***, el paràmetre **ideal_ecosistem** i guardarem l'arxiu sobre l'objecte **Arxiu_ideal** (Línea 2).

Seguidament, ens interessarà tenir els individus de **Arxiu_mitjana** i **Arxiu_ideal** en dues llistes, ja que caldrà calcular la diferència de quadrats entre ells. Per poder emmagatzemar els individus que tenim en **Arxiu_mitjana**, creem un bucle ***for*** (Línia 3-6) que recorrerà línia a línia **Arxiu_mitjana**, per cada iteració del bucle filtrarem pel caràcter "*" ja que serà el separador entre l'espècie a tractar i el nombre d'individus de les espècie, posteriorment els individus els guardarem sobre l'objecte **individu** (Línia 4). Finalment per cada iteració del bucle, acumularem els individus capturats sobre la llista **lst_individus_simulats** (Línia 5). El procés finalitzarà quan hem llegit **Arxiu_mitjana.**

Per obtenir els individus que hi ha en **Arxiu_ideal,** repetirem el mateix procés, amb la diferència que els individus capturats seran emmagatzemats en la llista **lst_individus_ideal** (Línia 9).

Ara sol ens queda recorre les dues llistes a la vegada utilitzant un bucle ***while*** (Línia 11-19). En el bucle realitzarem el següent procés: sempre que la **lst_individus_simulació** no estigui buida i que el valor que tenim capturat sigui major a zero (Línia 12), calcularem la diferència de quadrats (Línia 13-14). Quan capturem el valor, l'emmagatzemarem en la llista contenidora **def_quadrat_escenari** on tindrem per tots els anys estudiats en l'ecosistema real, la seva diferència de quadrats amb els valors calculats amb el simulador.

Per acabar el procés, sumarem els valors de **def_quadrat_escenari** i retornarem el valor de la diferència de quadrats per l'escenari estudiat.

3.5.2 Consideracions dels problemes apareguts en obtindre l'escenari ideal

La principal problemàtica a l'hora de realitzar l'script, ha estat a l'hora de recorre les dues llistes per obtenir els valors de la diferència de quadrats. Ja que, l'usuari expert en múltiples ocasions no té informació de l'espècie estudiada per cada un dels anys, i quan pretenem comparar amb els valors obtinguts en les simulacions, els quals si que tenen valor capturat per cada any d'estudi, provocava errors.

Per poder-ho solucionar, vam decidir que si per algun any d'estudi l'expert no tingues informació capturada, a l'hora de realitzar la diferència de quadrats, no es tingues en compte aquell any en el càlcul de l'error.

3.6 Execució de l'API de càlcul d'ecosistemes

Un cop tenim totes les funcions necessàries per poder generar les simulacions d'un escenari i obtenir l'error per l'ecosistema que estem estudiant, és hora de crear un script genèric que executi tots els anteriors script d'una forma seqüencial i neta.

Per poder realitzar aquest procés, hem implementat l'script **estadistic.py**, aquest script és l'executor de tot el procés. S'encarregarà d'executar de forma seqüencial tots els script implementats. Inicialment cridarà la funció ***crida_parser()*** de l'script **parser.py**, seguidament la funció ***crida_compiler()*** de l'script **compiler.py**, a continuació ***crida_process()*** de l'script **process.py** i finalment obtindrem l'error amb la funció ***crida_ideal()*** de l'script **Ideal.py**.

Un cop tenim centralitzat tot el procés en un script. Ja podem tractar el nostre sistema de càlcul d'ecosistemes, com una API de càlcul d'ecosistemes.

La missió d'aquesta API, és ser cridada quan sigui necessari des d'una aplicació exterior, ja que serà utilitzada com un mecanisme indispensable, per l'elecció de l'escenari més adient per l'expert. La crida es farà sobre la funció ***cerror()***, que contindrà tots els paràmetres necessaris per realitzar els càlculs.

A la Figura 21 mostrada a continuació podem veure un diagrama de flux del procés que realitzarà l'API.

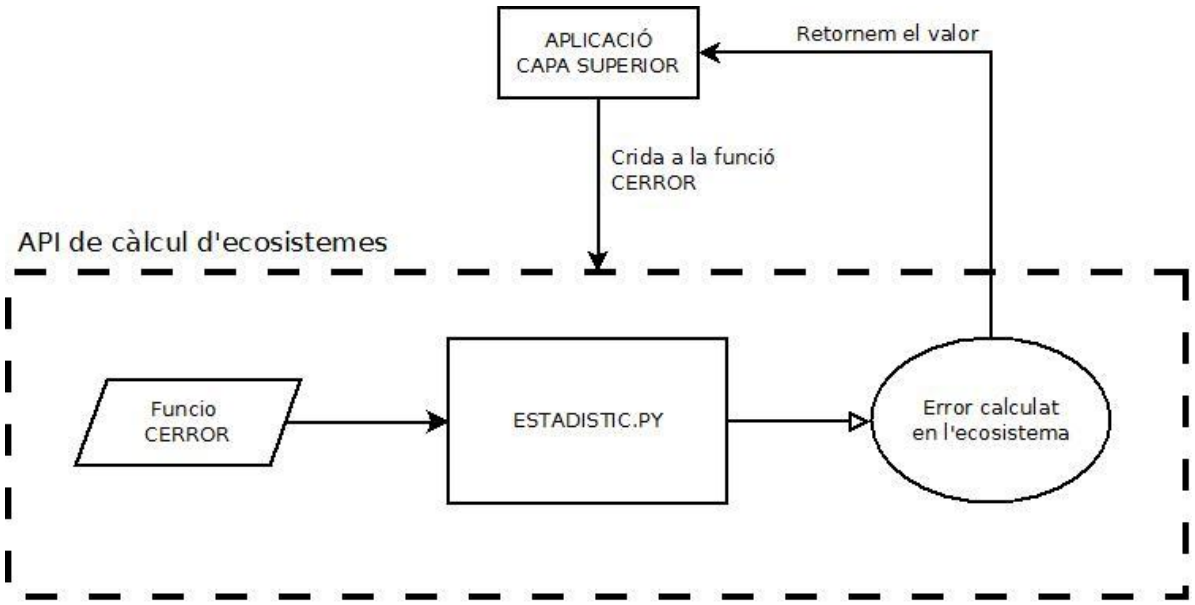

Figura 21: Diagrama d'execució de L'API de càlcul

3.6.1 Implementació de l'API de càlcul d'ecosistemes

Per poder executar l'script **estadistic.py**, serà necessari importar les funcions de cada un dels script implementats. D'aquesta forma, fent la crida sobre aquestes funcions amb els paràmetres necessaris, podem executar l'API sense problemes.

Les funcions importades són les següents: de **parser.py** importarem de la funció ***parser()***, de **compiler.py** importarem la funció ***com()***, de **process.py** importarem la funció ***process()*** i importarem la funció ***search_ideal()*** de l'script **Ideal.py**. A més a més, incorporarem una funció per eliminar la informació innecessària que generen els scripts, en els directoris **Pathin** i **Pathout**, que li direm ***rm()***.

Un cop tenim importades les funcions necessàries per realitzar el procés, podem definir la funció ***cerror()***, la qual requereix els següents paràmetres:

- **Steps:** El paràmetre contindrà cada quantes configuracions capturarem la informació de la simulació, per després poder-la tractar. Es tracta d'un valor enter.
- **Pathdat:** El paràmetre conte el directori on es troben les simulacions realitzades per el nostre simulador P-Sistema.
- **Scenaryname:** Nom de l'escenari que estem tractant, per poder discernir entre simulacions d'altres escenaris.
- **Ideal_ecosistem:** Directori on trobem l'arxiu amb les dades preses per l'expert en el medi.
- **Path_especies:** Directori on trobem l'arxiu editable d'espècies.
- **Path_estadistics:** Directori on guardem els arxius generats per el **process.py**.
- **Path_in:** Directori on guardem els arxius generats per el **parser.py**.
- **Path_out:** Directori on guardem els arxius generats per el **compiler.py**.

En la Figura 22 podem veure el diagrama de flux de **estadistic.py**, amb la seva ajuda podrem explicar de forma simple i entenedora els procés que realitzarà per cada escenari que estudiarem.

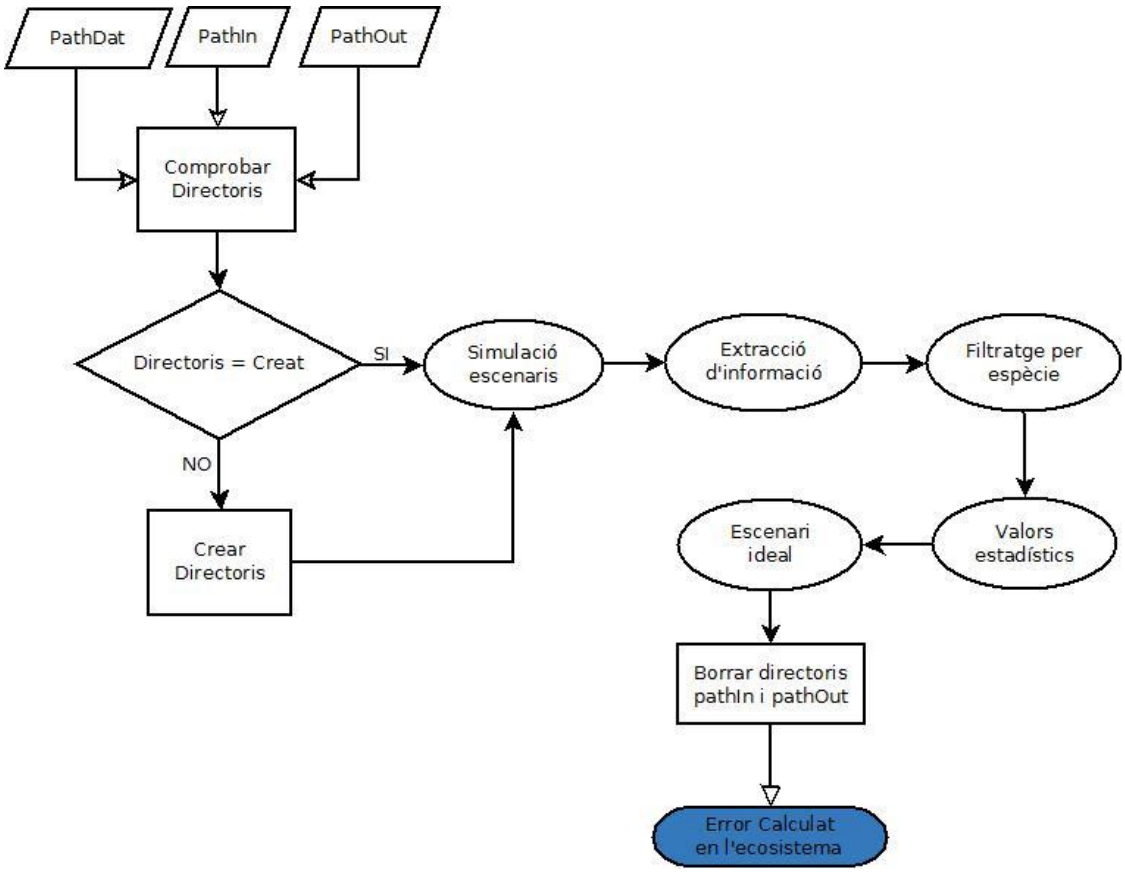

Figura 22: Diagrama de l'script estadistic

Inicialment comprovarem que els directori; **PathDat, PathIn**, **PathOut** existeixen. Si algun dels directoris **PathDat** o **PathIn** no existeix, és crearà de forma automàtica, ja que en aquest directoris emmagatzemarem els arxius resultants del parseig. Si **PathDat** no existeix, s'aturarà l'execució de l'API, ja que si no tenim escenaris a simular, no té sentit que continuï el procés.

Superat aquest filtre, executarem en cascada totes les funcions de l'API. Inicialment farem la crida sobre la funció ***parser()*** i capturarem els temps de processament de la mateixa. A l'acabar, farem el mateix per les funció ***com()***, ***process()*** i ***searh_ideal()***. Un cop finalitzat el procés, podrem veure en la consola del sistema operatiu Figura 23, els temps d'execució per cada un dels scripts que formen l'API i l'error obtingut per l'escenari estudiat.

```
[mtrigueros@maracas Llibreria_trito]$ ./estadistic.py
El directori file_in s'ha creat
El directori file_out s'ha creat

 Extracting STEPS data...
 Extracting STEPS data... OK

 Compiling SPECIES...
 Compiling SPECIES... OK

 Calculating statistics...
 Calculating statistics... OK

 Calculating ideal...
 Calculating ideal... OK

 Calculating ideal...
 Calculating ideal... OK
------------------------------------ Procés de parseig Finalitzat! ------------------------------------

 El temps en realitzar parser és de: 6.019208 segons
 El temps en realitzar compiler és de: 0.947708 segons
 El temps en realitzar process és de: 0.005140 segons
 El temps en realitzar l'ideal és de: 0.000311 segons
 Els temps TOTAL en realitzar el parseig és de: 6.972367 segons

 Error calculat: 1342782872.455875
-----------------------------------------------------------------------------------------------------
[mtrigueros@maracas Llibreria_trito]$
```

Figura 23: Execució de L'API de càlcul

3.6.2 Solució dels problemes derivats de la implementació de l'API de càlcul d'ecosistemes

La principal problemàtica a l'hora d'implementar l'script, va ser; decidir com estructurar el sistema de funcions encarregat de realitzar les crides als scripts implementats en el projecte, tenir clar els arguments necessaris que requeria cada funció per poder ser executada i que tot el procés fos fluït, ja que és molt important que tot el procés és realitzi amb el menor temps possible.

Ràpidament ens vam adonar, que Python tracta les funcions de forma molt similar que C++, degut a la nostra experiència prèvia amb aquest llenguatge de programació, vam solucionar la problemàtica inicial que sens presentava i hem obtingut un rendiment molt bo en la obtenció de la diferència de quadrats entre l'escenari simulat i el observat (detallat en la part experimental del projecte).

Capítol 4

4. Experimentació i resultats

L'objectiu de l'experimentació és estudiar els recursos necessaris en realitzar el procés tant de simulació d'escenaris com de parseig i ajust dels valors dels paràmetres. A més a més, analitzarem els respectius temps d'execució per una determinada arquitectura de computador i els recursos necessaris del sistema per emmagatzemar la informació extreta.

En primer lloc, es descriu l'entorn d'experimentació en el que s'han realitzat les proves. En segon lloc, es descriuen els temps de processament requerits pels scripts o funcions principals de la nostra API, segons el nombre de simulacions que es defineixen per escenari. En tercer lloc, es descriu la quantitat de recursos necessaris per emmagatzemar i processar els resultats obtinguts. Per últim, s'analitzen els resultats obtinguts.

4.1 Entorn d'experimentació

La informació més important de l'entorn en que s'ha realitzat l'experimentació és la següent:

- **Memòria del node:** Quantitat de memòria principal de la que disposa el node. Aquest paràmetre és fonamental ja que si no es defineix correctament el Plinguacore no crearà les simulacions de l'escenari. En el nostre entorn és de 4GB per node.
- **Potència de còmput del node:** L'entorn d'experimentació disposa de 24 màquines amb uns processadors Intel Core 2 Quad a 2.4GHz.
- **Sistema operatiu:** La versió del sistema operatiu que s'ha utilitzat ha estat una redistribució de Linux anomenada Rox 5.3 .
- **Entorn on es realitzen simulacions:** Les simulacions son executades en el servidor del Cluster Maracas[4].
- **Entorn on s'executa el mòdul de càlcul implementat:** El procés d'execució és pot dur a terme tant en local com en el front-end d'un entorn distribuït, Cluster, Multicluster,etc.

4.2 Temps de processament per script

Volem avaluar l'eficiència de l'API implementada en quant al processament dels resultats generats pel simulador PlinguaCore. Per tal de realitzar aquestes proves hem analitzat els temps de les principals funcions de la nostra API en el processament dels resultats obtinguts en la simulació d'un escenari concret de l'ecosistema del Tritó Pirinenc[13] amb quatre configuracions diferents, on variem la quantitat de les simulacions que s'han de fer de l'escenari per tal de normalitzar els resultats.

La Taula 1 mostra els temps d'execució respectius per cada script i el temps total en realitzar tot el procés.

Scrips	Temps/s 1 simulació	Temps/s 30 simulacions	Temps/s 50 simulacions	Temps/s 100 simulacions
Simulació d'escenaris	5,907024	185,0325	299,0132	557,3058
Extreure info	0,120661	3,695955	6,078395	12,066118
Filtrar espécies	0,018956	0,56895	0,948051	1,896106
Fitxer estadistics	0	0,000206	0,0005155	0,00989
Comput ideal	0	0,0002329	0,0002119	0,0002939
Temps Total	**6,046641**	**189,2978439**	**306,0403734**	**571,2782079**

Taula 1: Taula de processament d'scripts en segons

Com es pot veure en la taula, a mesura que incrementem el nombre de simulacions d'un mateix escenari creix de forma lineal el temps en realitzar el procés. Tot i així, el temps necessari en executar la majoria de les funcions de la API és ínfim, essent del ordre dels nanosegons en les funcions de càlcul estadístic i comprovació amb les dades reals, de milisegons en el filtratge d'espècies i d'uns pocs segons en la extracció d'informació. Es pot observar també que el procés en que més temps s'inverteix amb diferència, és en el procés de simulació, que creix ràpidament amb el nombre de simulacions, arribant a una durada de l'orde de la desena de minuts en el pitjor dels casos, és a dir, amb 100 simulacions.

.

En la Figura 23 es pot observar de forma molt més clara el increment en temps d'execució entre la simulació d'escenaris respecte el primer script de l'API de càlcul encarregat de treballar amb totes les simulacions generades per el PlinguaCore i posteriorment parsejar'ls amb la intenció de queda'ns solament amb la informació que realment l'interessa a l'expert.

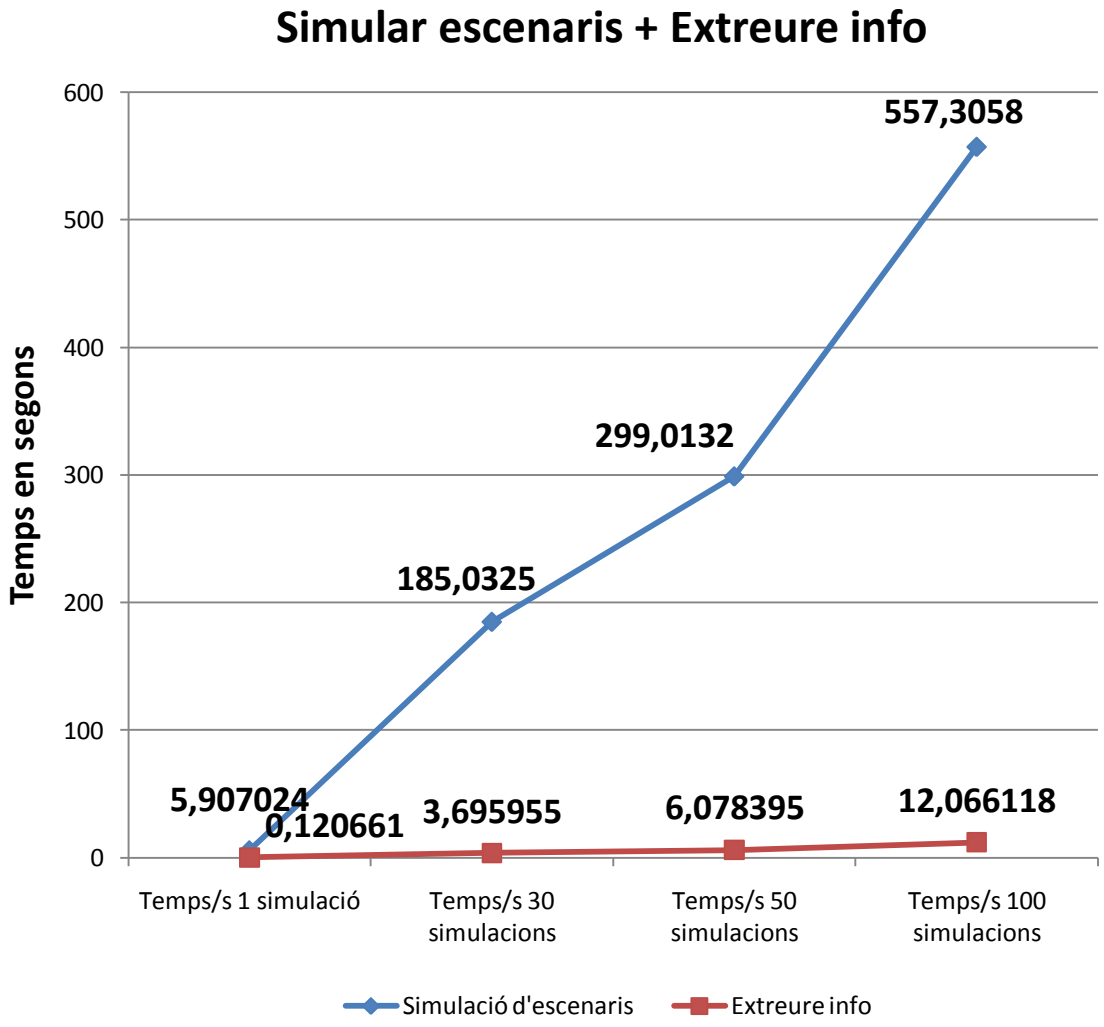

Figura 24: Gràfica de simulacions respecte primer parseig

Pel que fa a l'eficiència obtinguda en la API implementada en el projecte, es força bona, ja que en el pitjor dels casos el temps que triga en fer el primer parseig de les dades extretes del PlinguaCore es de l'ordre dels 12 segons.

Degut a la diferència en l'escala de temps en obtenir les dades del parseig realitzat pel **parser.py** respecte a la resta d'scripts de l'API (de l'ordre dels nanosegons). Ens ha estat necessari observar el creixement lineal de l'API, utilitzant una nova gràfica per tal que sigui més entenedor i no sens solapin les dades.

La Figura 24, mostra el temps necessari per configuració en executar els script de l'API exceptuant el **parser.py**.

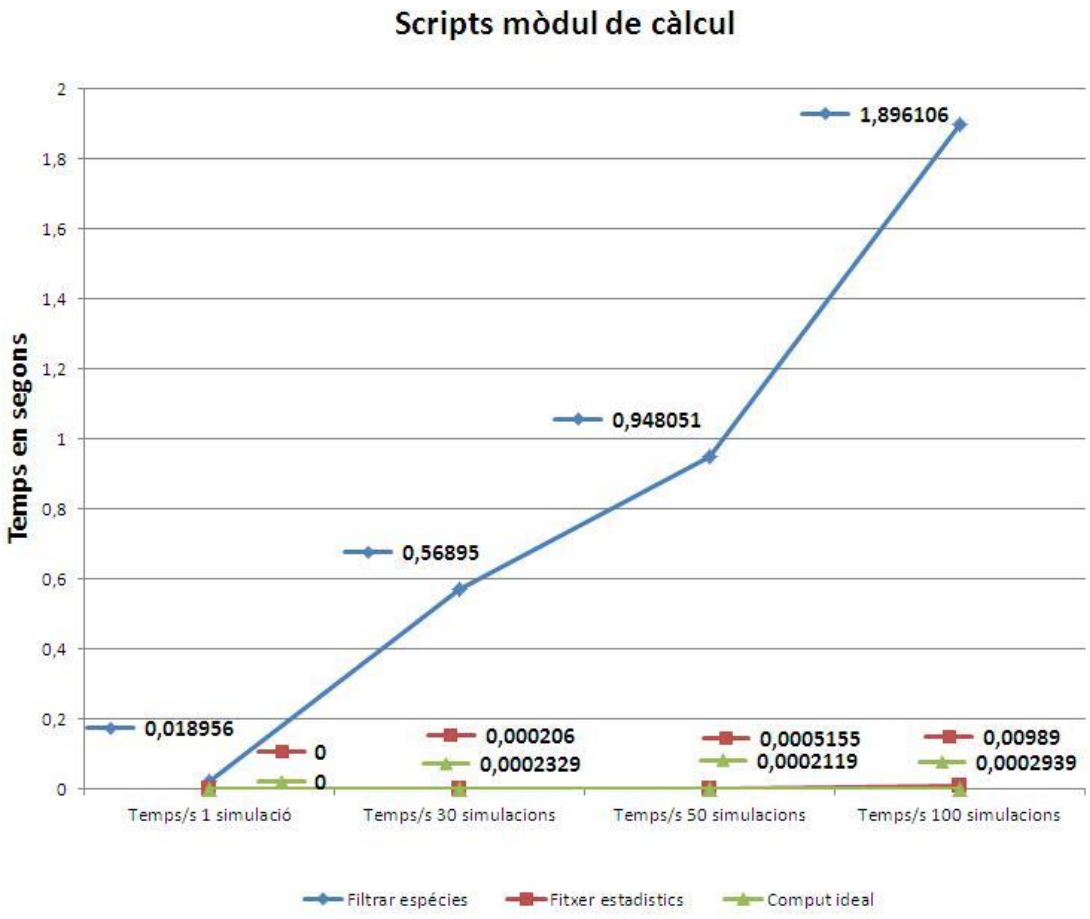

Figura 25: Resta d'scripts de l'API de càlcul

4.3 Recursos consumits per el procés

Un cop realitzem el procés de simulació d'un escenari i posteriorment l'execució de la nostra API de càlcul, ens hem adonat que generem gran quantitat d'informació, en primer lloc degut a les simulacions realitzades per el PlinguaCore i posteriorment per tot el procés de parseig que realitza el mòdul de càlcul implementat.

Per aquest motiu, l'equip on és realitza el procés ha de tenir els recursos suficients per poder realitzar el procés sense problemes ja que tindrà d'emmagatzemar molta informació

Com hem fet en el punt (4.2) treballarem amb quatre configuracions de calibratge diferents mostrades en la Taula 3 per realitzar les nostres proves.

Scrips	Recursos en generar 1 fitxer	Recursos en generar 30 fitxers	Recursos en generar 50 fitxers	Recursos en generar 100 fitxers
Simulació d'escenaris	3,4	102	169,9	340
Extreure info	0,04473	1,4	2,3	4,6
Filtrar espécies	0,00026	0,00791	0,01318	0,02637
Fitxer estadistics	0	0,00069	0,00069	0,0007
Recursos Totals/MB	**3,44499**	**103,4086**	**172,21387**	**344,62707**

Taula 2: Taula de recursos per script en MBytes

En la Figura 25 podem veure el nombre de recursos totals, un cop executat el nostre mòdul de càlcul per cadascuna de les configuracions estudiades per l'expert.

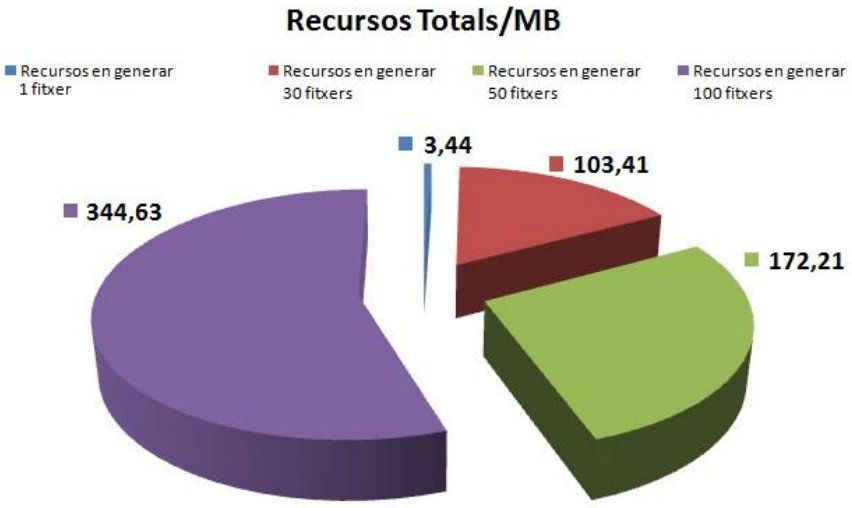

Figura 26: Gràfica recursos consumits en el procés

4.4 Anàlisis dels resultats obtinguts

En aquest punt es mostren els anàlisis dels resultats obtinguts tant en el temps requerit en el processament dels resultats, com en la quantitat de recursos d'emmagatzematge necessaris segons la quantitat de simulacions que l'usuari expert decideix fer per escenari.

Temps en processar els scripts:

Com a conseqüència de l'estudi realitzat en el punt (4.2), podem dir, que la variació de temps entre les quatre diferents configuracions elegides per realitza les proves, es pràcticament lineal i força bo, ja que es poden prendre decisions amb gran rapidesa, de l'ordre de la desena de minuts en el pitjor dels casos, és a dir, amb 100 simulacions d'un escenari, obtindríem els resultats tant sols en 13 segons.

Cal dir, que en el procés d'obtenció de simulacions d'un ecosistema, procés realitzat per el simulador de ecosistemes PlinguaCore, no es pot dir el mateix, ja que requereix de molt més temps en crear les simulacions de l'escenari.

En les quatre diferents configuracions elegides com a exemple, a mesurar que l'usuari expert necessita més simulacions per obtenir resultats més fiables, els temps d'execució creixen moltíssim i això provoca que el procés sigui més lent. Degut a la experiència obtinguda en el processament dels escenaris i en l'obtenció de l'escenari més proper al comportament real, podem dir, que una xifra encertada de simulacions si volem obtenir uns resultats fiables, seria de 100 simulacions.

Si que el temps en realitzar el procés és més gran, però ens estem movem al voltant de 15 segons/escenari, xifra acceptable per realitzar un bon estudi.

Recursos necessaris dels equips:

Per poder realitzar la simulació de l'escenari més l'execució de l'API de càlcul implementat en el projecte, requereix que els equips on es realitzin els experiments tinguin els recursos necessaris per poder emmagatzemar tota la informació que generem.

Ja que cal creuar les dades obtingudes de totes les simulacions abans de fer la comparació amb els resultats reals. Contra més simulacions millor serà la precisió, però per contra, requerirem de molts més recursos d'emmagatzemament.

Quan el nombre d'escenaris a tractar i el nombre de simulacions es molt elevat, l'espai necessari pot superar fàcilment l'espai d'un a maquina convencional.

Una solució a aquest problema és eliminar tots els arxius amb els resultats entremitjos en cada nova simulació i emmagatzemar les dades obtingudes en una base de dades. De forma que un cop processat un escenari, tota la informació generada es pugui eliminar de l'espai d'emmagatzemament local.

Capítol 5

5. Conclusions i treball futur

L'objectiu plantejat inicialment en aquest projecte consistia en l'ús d'un sistema de parseig d'arxius. Aquest arxius ens son proporcionats després de la simulació d'un escenari d'un ecosistema processat per el simulador d'ecosistemes PlinguaCore[5].

Aquest objectiu no només s'ha complert sinó que s'ha millorar el sistema de creació de simulacions generades per el PlinguaCore, modelant un script Python que simula cada escenari tants cops com l'usuari ecòleg indica.

Gràcies a la implementació del sistema de parseig dissenyat en el present projecte, donat una sèrie d'escenaris del mateix ecosistema hem estat capaços de decidir quin és l'ecosistema que considera l'expert que més s'apropa al model considerat ideal.

A més a més, la API implementada en aquest projecte s'ha integrat en el projecte del Abert Agraz[14], per realitzar aquesta operació ha estat necessari reconvertir els nostres scripts perquè actuessin com una API , la qual seria cridada quan fos necessari.

En conclusió, hem implementat una API que facilita la feina del processament dels resultats del PlinguaCore. Aquesta API, és multiplataforma i és pot fer servir tant per calibrar un model com per processar futures simulacions en que es vulgui predir el comportament dels ecosistemes, etc.

S'ha integrat la API en un sistema de calibratge i se n'ha analitzat el seu rendiment. S'ha provat que les funcions de la API tenen un rendiment molt bo ja que s'executen en molt poc temps. Això permet poder escalar el problema i treballar amb grans models que requereixen una gran quantitat de recursos tant de còmput com d'espai de disc .

Un cop finalitzat el projecte podria plantejar algunes línies de treball futur per tal de millorar el present treball:

- Per millorar l'eficiència del sistema, seria interessant modificar el simulador d'ecosistemes PlinguaCore de forma que els resultats obtinguts de cada simulació fossin solament els que necessita l'expert estudiar. D'aquesta forma ens podríem estalviar el procés de filtratge centrat tant sols en obtenir la informació interessant per l'expert.

- Implementar un sistema perquè un cop obtinguts els resultats del mòdul de càlcul fossin emmagatzemats en una base de dades on l'expert pugues consultar els resultats de forma ràpida, ja que estudiarà múltiples escenaris per cada ecosistema.
- És podria optimitzar l'API, afegint en el fitxer editable, de quines membranes li interessa a l'usuari expert analitzar la informació. Realitzant aquest procés, obtindrem resultats més específics.

Bibliografía

[1] Python Programming Language – Official Website, http://www.python.org

[2] Tesis Perez Urtado, Dessarrollo y aplicacions de un entorno de programacion para computacion Celular: P-Lingua

[3] Raúl González Duque, Python para todos, http://www.mundogeek.net/tutorial-python/

[4] Informació sobre el Maracas i el sistema de cues, http://www.gcd.udl.cat

[5] http://www.p-lingua.org/wiki/index.php/PLinguaCore.

[6] http://www.p-lingua.org/wiki/index.php/Main Page.

[7] Päun, G, Computing with membranes, Journal of Computer Systems Science 61, 108-143, 1998.

[8] http://en.wikipedia.org/wiki/P_system

[9] Cardona M., Colomer M.A., Pérez-Jiméenez M.J., Sanuy D., Margalida A., Modelling ecosystems using P Systems: The Bcarded Vulture, a case study., Lecture Notes in Computer Science 5391, 137-156, 2009

[10] Colomer, M.A., Margalida, A. & Sanuy, D. & Pérez-Jiménez, M.J., A bioinspired computing model as a new tool for modeling ecosystems: the avian scavengers as a case study., Ecological Modelling 222, 33-47, 2011.

[11] Cardona, M., Colomer M.A., Margalida A., Pérez-Hurtado I., Pérez Jiménez M.J., Sanuy D., A P-System based model of an ecosystem of some scavenger birds., Lecture Notes in Computer Science 5957, 182-195, 2010.

[12] Margalida, A., Colomer, M.A. & Sanuy, D., Can wild ungulate carcasses provide enough biomass to maintain avian scavenger populations? An empirical assessment using a bio-inspired computational model., PLoS One 6, e20248, 2011.

[13] Colomer M.A., Montoti A., García E. & Fondevilla C., A computational model to explain anual uctuations and extinction risk due to climate change related waterow in a Calotriton asper population., EPIC – Environment & Pyrenees International Conference, Universidad de Navarra, 2011.

[14] TFC Albert Agraz., Paralelización de un algoritmo de calibrado para el moldeado de ecosistemas.

[15] Web on trobem informació sobre Bash script: http://es.wikipedia.org/wiki/Bash.

[16] Llenguatge Ruby: http://www.ruby-lang.org/es/.

[17] Llenguatge Perl: http://es.wikipedia.org/wiki/Perl.

[18] Llenguatge Groovy: http://en.wikipedia.org/wiki/Groovy_(programming_language).

[19] Llenguatge Lua: http://www.lua.org/.

[20] Llibreria per realitzar crides a subprocessos: http://docs.python.org/library/subprocess.html.